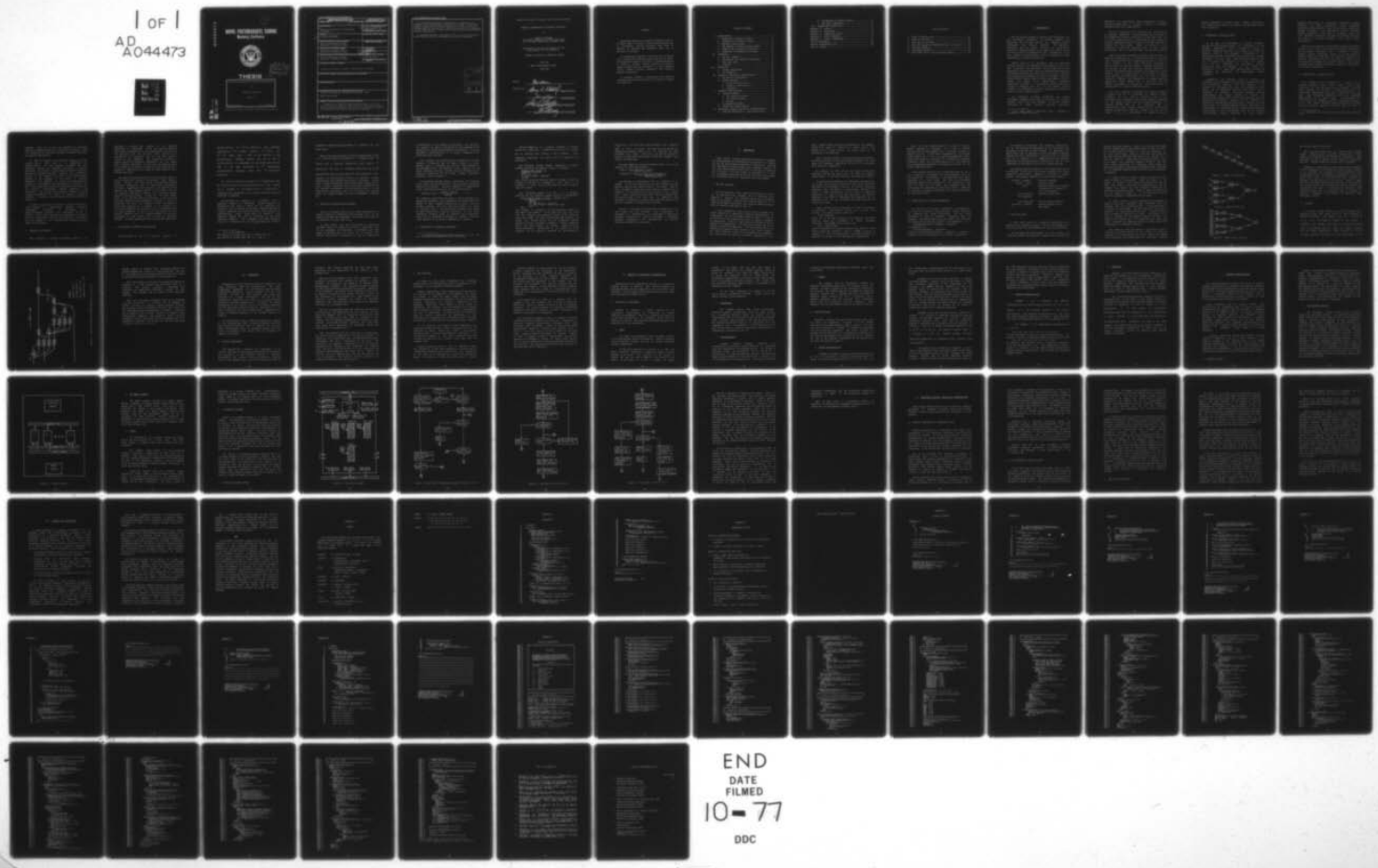
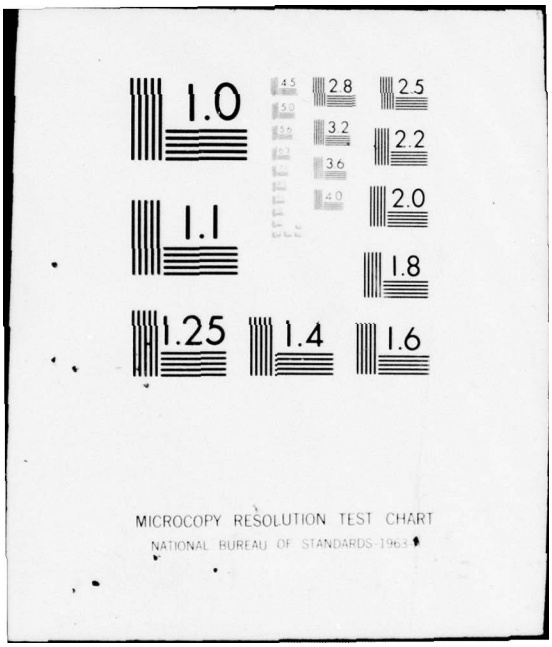
AD-A044 473 NAVAL POSTGRADUATE SCHOOL MONTEREY CALIF F/G 9/2
PARALLEL PROCESSING OF RECURSIVE FUNCTIONS.(U)
JUN 77 F BURKHEAD

UNCLASSIFIED NL

1 OF 1
AD A044473



END
DATE FILMED
10-77
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

Parallel Processing of Recursive Functions

by

Franklin Burkhead

June 1977

Thesis Advisor: G. A. Kildall

Approved for public release; distribution unlimited

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle) (6) Parallel Processing of Recursive Functions,	5. TYPE OF REPORT & PERIOD COVERED (9) Master's Thesis; June 1977
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) (10) Franklin/Burkhead	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940	12. REPORT DATE (11) June 1977
	13. NUMBER OF PAGES (12) 88 pp.
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Naval Postgraduate School Monterey, California 93940	15. SECURITY CLASS. (of this report) Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

parallel processing, recursive functions, LISP,

data flow graphs

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

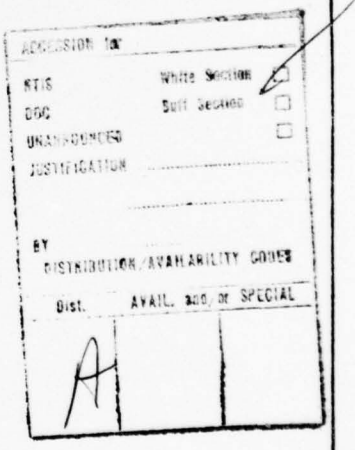
Algorithms defined as recursive functions, such as in "pure" LISP, are shown to have structure sufficient to distinguish between processes which must be executed in sequence and processes which may be executed in parallel. → next page

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

An interpreter program is presented for executing LISP programs and simultaneously computing the number of processors needed at each step of program execution in order to achieve optimum parallel processing. Sample program runs are presented to show speed-up ratios between strictly sequential and optimally parallel executions.

A possible hardware organization for a parallel processing system derived from the interpreter program is presented.



Approved for public release; distribution unlimited

PARALLEL PROCESSING OF RECURSIVE FUNCTIONS

by

Franklin Burkhead
Lieutenant Commander, United States Navy
B.S., United States Naval Academy, 1966

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

June 1977

Author: F Burkhead

Approved by: Gary A. Kildall
Thesis Advisor

Daniel Davis
Second Reader

Chairman, Computer Science Department

Dean of Information and Policy Sciences

ABSTRACT

Algorithms defined as recursive functions, such as in "pure" LISP, are shown to have structure sufficient to distinguish between processes which must be executed in sequence and processes which may be executed in parallel.

An interpreter program is presented for executing LISP programs and simultaneously computing the number of processors needed at each step of program execution in order to achieve optimum parallel processing. Sample program runs are presented to show speed-up ratios between strictly sequential and optimally parallel executions.

A possible hardware organization for a parallel processing system derived from the interpreter program is presented.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

Both the growing volume of business data processing and the increasing complexity of problems emerging from scientific and military research are demanding greater efficiency[1] from computer systems. The efficiency attainable from the latest versions of traditional, sequential machines appears to be approaching limits set by the final speed of electromagnetic wave propagation, while more costly, non-traditional, parallel machines have been encumbered by software complexities.

Because much of the processing done by sequential machines consists of independent sub-processes, it has long been recognized that efficiency could be improved if these independent sub-processes could be performed simultaneously. Considerable progress has been made in this area by building more sophisticated systems from basic Von Neumann machines. Multiprocessor systems enable multiple processors to share a common core memory while simultaneoulsy processing independent programs. A distributed system represents a network of computers, each with its own memory, working together on independent (or nearly independent) programs in order to solve a common problem.

But individual programs themselves may contain independent segments which could be executed in parallel. Machines which have been designed to perform parallel executions of single programs represent significant

---

[1] As used herein, efficiency means a measure of throughput or execution speed.

departures in organization from conventional machines. Thurber and Wald [Ref. 1] provide a survey of parallel processors and their organizations.

There is a reason for the unconventional architectures of parallel processors. Single programs may be thought of as mathematical functions which map a single input data set to a single output data set. This means that the results of parallel computations must eventually be brought together to produce the final output. Hence, the actions of the components of a parallel processor must be more tightly coordinated than in a multiprocessor or distributed system.

There are several reasons why parallel machines have not achieved widespread use. Certainly one reason is that many users remain satisfied with sequential machines as long as they continue to meet their efficiency requirements. Another reason is that the parallel machines developed so far require an additional degree of software complexity in order to distinguish between parallel and sequential tasks. Hardware cost is another reason. In the past, the cost of logic elements was much greater than the cost of memory elements. Memories were designed to be accessed through a single port. These factors were in line with the Von Neumann principles of sequential, centralized control of computations and linearly organized memory.

But now, advances in technology are making it possible to define a new set of principles for computer design. Glushkov, et. al., [Ref. 2] present a set of five principles, quite different from the Von Neumann principles, for the design of what they call recursive machines. The advances already made by LSI technology make the possibilities seem endless. Manufacturers are currently producing single-chip computers (memory and CPU on one chip). It is not inconceivable to imagine an array of

bipolar processors on a single chip. Memory technologies are improving too, making it possible to access data faster and in parallel.

A. DEVELOPING A PARALLEL SYSTEM

In the past the development of parallel processor systems has been characterized by the development of the hardware organization first, followed by efforts to implement compatible software. As an example, the ILLIAC IV computer [Ref. 3] was designed to capitalize on the parallelism inherent in problems where the data is naturally structured in array form. The processing elements, each with 2K of memory, are organized into four 8 x 8 arrays. Kuck [Ref. 4] discusses the programming language Tranquility which was designed for the ILLIAC IV. Tranquility is an algol-like language which provides the programmer with sequential and simultaneous control statements.

Ramamoorthy and Gonzales [Ref. 5] suggest two approaches to the problem of recognizing program tasks which can be executed in parallel. The first approach is to provide the programmer with tools, like Tranquility, which enable him to explicitly indicate tasks which can be processed in parallel. The second approach involves preprocessing the source program to analyze the relationships between tasks and thus determine what parallel processing is possible. Lamport [Ref. 6] presents two methods for enabling parallel execution of Fortran DO loops. Keller [Ref. 7] discusses methods whereby processors can "look-ahead" to a limited number of sequentially organized instructions to find instructions that can be executed "out-of-order" without affecting the final outcome.

Research into methods of recognizing independent program segments, and hence parallelism, within sequential algorithms, seems worthwhile since it may permit established program libraries to be efficiently utilized on future parallel processors. Stone [Ref. 8] however, points out that efficient algorithms designed for parallel execution may prove to be quite different from their serial counterparts.

Based on the work done in developing parallel processing systems so far, and on the recent and predicted advances in LSI technology, a reasonable way to implement a general purpose parallel processing system is to first develop a software system for describing the parallel execution of computer algorithms and then to organize the hardware so as to physically implement the software system. This thesis considers such a software system and suggests an approach to the hardware organization.

B. SOFTWARE FOR A PARALLEL SYSTEM

The software system considered is a subset of an existing language, LISP, whose syntax allows easy recognition of parallel tasks within a program. In "pure" LISP, programs are defined as recursive functions of conditional expressions which act on ordered sets of input data. When evaluating algorithms which are described functionally, as in "pure" LISP,[2] the procedure is to first evaluate the arguments and then to apply the function. It is this simple procedure which differentiates between what can be done in parallel and what must be done in

---

[2] Henceforth, the term LISP will be used to mean "pure" LISP.

sequence. That is to say, the arguments to a function represent processes which can be executed in parallel, while the composition of functions represent processes which must be executed sequentially.

In order to "reveal" the parallelism inherent in a LISP program and to show that it is recognizable at execution-time, the LISP function evalquote2 has been developed. Section III describes evalquote2 in detail. Evalquote2 is similar to the universal function evalquote. Evalquote is called a universal function (or interpreter) because it can compute the result of any LISP function applied to its arguments if the result is defined. Evalquote2 also computes the result of any LISP function applied to its arguments, and additionally, it monitors the data flow graph which describes graphically the sequential and parallel relationships between executing LISP primitives. The output from evalquote2 includes a list of integers representing the number of separate processors required to optimize parallel processing at each stage of execution.

In order to postulate the effect of running non-trivial, LISP programs in a hypothetical parallel processing environment, evalquote2 is implemented by a LISP-metalanguage translator and interpreter written in Algol-W. This Algol-W program will henceforth be referred to as the interpreter. Section IV explains the interpreter and the results it has obtained from processing several sample LISP programs.

C. MEASURING PARALLELISM

When proposing a parallel processing system it is

necessary to provide some measure of the expected improvement in efficiency. Stone [8] uses the speed-up ratio which is defined for a given algorithm as the ratio of the execution time for the best serial version of the algorithm to the execution time for the best parallel version of the algorithm. The interpreter provides a similar measure of efficiency improvement for the sample LISP programs evaluated. In this case, the speed-up ratio is the ratio of the number of execution steps required for a sequential execution to the number of stages required for a parallel execution.

The sample programs analyzed by the interpreter were not chosen because they generate particularly large speed-up ratios. Rather, they were chosen as "typical" programs offering a reasonable blend of conditional expressions, functional composition, and recursion. The speed-up ratios computed for these programs provide a very limited view of the improved efficiency possible with a general purpose parallel processing system. ILLIAC IV, the most widely known of the existing parallel processors, is call an array processor because it was developed to process a class of algorithms for which the speed-up ratios are enormous. Matrix multiplication is an example of an operation for which large speed-ups are possible, and it is included among the sample programs. In order to gain some insight into the results expected from the sample programs, the remaining paragraphs of this section will discuss the speed-ups possible in matrix multiplication and summation algorithms.

D. THE EXAMPLE OF MATRIX MULTIPLICATION

Multiplication of two $n \times n$ matrices requires n^3

multiplications and $n^2(n-1)$ additions. When performed sequentially, this process requires $n^3 + n^2(n-1)$ (or $2n^3 - n^2$) steps, where a step is one addition or one multiplication. Observe, however, that all of the n^3 multiplications are independent of one another and could be done in one step consisting of n^3 simultaneous multiplications (assuming there were n^3 multipliers available).

The $n^2(n-1)$ additions represent the n^2 summations, each of n products from the multiplications, which will produce the n^2 elements of the product matrix. Certainly the n^2 summations are independent of one another and hence could be performed in parallel.

Now consider the summation of n elements. Such a summation requires n-1 additions. Because addition is associative, the order in which the n-1 additions are preformed will not affect the outcome. Because addition is a binary operation, the summation process can be started by simultaneously adding $\lfloor n/2 \rfloor$ pairs of addends.[3] The summation process can then be reapplied to the $\lceil n/2 \rceil$ remaining elements. This procedure will still require a total of n-1 additions, but only $\lceil \log_2 n \rceil$ steps are required

---

[3] For a real number x,
$\lceil x \rceil$ denotes an integer such that $x \leq \lceil x \rceil < x+1$, and
$\lfloor x \rfloor$ denotes an integer such that $x-1 < \lfloor x \rfloor \leq x$.

(assuming a minimum of $\lfloor n/2 \rfloor$ adders are available for the first step).

Hence, the total process of matrix multiplication of two n x n matrices could be performed in $1 + \lceil \log_2 n \rceil$ steps. Of course, such a parallel computation would require n^3 multipliers for step 1, $n^2(\lfloor n/2 \rfloor)$ adders for step 2, and approximately half as many adders for each successive step. Consider the multiplication of two 8 x 8 matrices. If done sequentially, this process would require 960 steps. If done with optimum paralleling, this process would require only 4 steps! Hence, a speed-up ratio of 240 could be achieved by 512 parallel multipliers and 256 parallel adders. A LISP program which multiplies two 4 x 4 matrices is included among the sample programs and will be discussed in Section IV.

E. SPEED-UP FOR ASSOCIATIVE PROCESSES

There is a general result for the speed-up possible in a process composed of associative sub-processes such as the summation process just discussed. To develop this result it is necessary to define some terms.

As used herein, the term "primitive" (or "primitive process") refers to a member of the set of operations that can be performed by a processor. A processor is an agent (human or machine) which can carry out a process. The process may be just a primitive, or it may be a composition

of primitives. In the example of multiplying two matrices, the process of matrix multiplication was composed of the primitives for scalar multiplication and addition. For some processors, multiplication is a process composed of the primitives shift and add.

The operands for primitives may be referred to as data elements. A primitive may be unary, binary, or n-ary, meaning that it processes one, two, or n data elements in one processing step. The physical action implied by the terms primitive process, process, and data elements can be described abstractly by the terms initial functions, functions, and set elements.

The following general formula represents the speed-up ratio which can be achieved by an ideal parallel processing system[4] for a general process composed of associative, n-ary, primitives acting on a set of N data elements.

$$\text{speed-up ratio} = \frac{\lceil (N-1) \;/\; (n-1) \rceil}{\lfloor \log_n N \rfloor} \quad , \quad n \geq 2$$

The numerator represents the number of steps required for a sequential execution. Each sequential step would reduce the number of data elements remaining by n-1 until n or fewer elements remained. The final step would reduce the number of elements to one. The denominator represents the number of steps required for a parallel execution. At each step, successive n-tuples would be operated upon in parallel. Brent [Ref. 9] provides an analysis of parallel execution times possible for arithmetic expressions in general.

F. COMPARISON OF SUMMATION ALGORITHMS

---

[4] An ideal parallel processing system is one that has all the primitive processors it will ever need.

Because summation is a process composed of binary, associative additions, a speed-up ratio of $(N-1) / |\log_2 N|$ can be achieved when summing N data elements. Three summation algorithms, one serial and two parallel, are considered.

The following program segment represents a typical FORTRAN subroutine for summing a vector of integers.

```
      FUNCTION SUM (INTGRS, N)
      DIMENSION INTGRS(N)
      ISUM = 0
      DO 1 I = 1, N
1        ISUM = ISUM + INTGRS(I)
      RETURN
```

This serial algorithm actually requires N steps. Since the FORTRAN DO loop will be processed at least once, it is necessary to allow for the case where the vector contains only one integer.

The following segment from Ref. 4 is a parallel algorithm for the summation process written in TRANQUILITY.

```
BEGIN INTEGER ARRAY A[0:255];  INTEGER I,J,K,;
    FOR (K) SEQ (0,...,7) DO
        BEGIN
        J ← 2↑K;
        FOR (I) SIM (0,...,255) DO
            A[I] ← A[I] + A[(I+J) MOD 256]
        END;
END:
```

This segment is designed to use 256 processor elements to sum exactly 256 elements. If the input data set has less than 256 elements, the remaining elements of the array are given zero values. If there are more than 256 elements the extras are folded across the 256 processing element memories, and each processor performs a serial summation before the above segment is invoked. Referring to the result developed in sub-section D, the number of steps required for the parallel summation of 256 elements is

$\log_2 256$ or 8. The outer FOR loop represents this sequence (SEQ) of 8 (0,...,7) steps. The inner FOR loop causes the simutaneous execution (SIM) of the '+' primitive by each processor (0,...,255). This inner loop will be executed a total of 8 times after which each processing element will contain the final sum.

The last summation algorithm presented here is in the syntax of the LISP metalanguage.

```
sum[a] = [null[cdr[a]] → car[a];
          T → sum[reduce[a]]]

reduce[a] = [null[a] → NIL; null[cdr[a]] → a;
             T → cons[add[car[a]; cadr[a]];
                      reduce[cddr[a]]]]
```

The variable 'a' represents a list of integers to be summed. If the list contains two or more integers, the sum function calls on the reduce function. The reduce function adds successive integer pairs and returns a reduced list of integers. The sum function is then applied to the reduced list. This process continues until the list contains only one integer which is the final sum. By computing these arguments in parallel the above algorithm will generate the sum of N elements in $\lceil \log_2 N \rceil$ addition steps.

Section II provides some background information on data flow graphs and LISP programs. This information is necessary for understanding the development of evalquote2 discussed in Section III. Section V proposes a "skeletal" hardware organization for implementing the parallel execution described by evalquote2.

II. BACKGROUND

This section contains background information necessary for understanding the development of Evalquote2 in Section III and the test programs discussed in Section IV. Included is a discussion of LISP concepts and a modified version of the LISP metalanguage syntax which is used for the programs in this thesis. Data flow graphs are explained as a tool for recognizing parallelism, and the g-vector is introduced as a notational device for describing data flow graphs.

A. THE LISP LANGUAGE

The LISP language is best described by Section I of Ref. 10. An overview of the language will be provided here. Appendix A contains the language syntax for the programs used in this thesis. Because these programs were run on a S/360 using EBCDIC characters, the notation differs slightly from the notation published in Ref. 10.

A LISP program is a LISP function and an argument list whose elements are S-expressions (symbolic expressions). An S-expression can be one symbol called an atom, or it can be an ordered list of S-expressions which is usually delimited by parentheses. There are three primitive functions used to manipulate S-expressions. The CAR function gives the first element within an S-expression. The CDR function gives the S-expression remaining after removal of the first element. The CONS functions takes two S-expressions and produces a new S-expression by inserting the first S-expression as the

first element within the second S-expression. For example, CAR<(A B C)> gives A, CDR<(A B C)> gives (B C), and CONS<A; (B C)> gives (A B C). An empty list, (), is equivalent to the atom NIL.

There are two primitive functions which are predicates. EQ gives the atom T if its two arguments represent the same atom, or F otherwise. ATOM gives the atom T if its argument is an atom, or F otherwise.

The version of LISP used in this thesis includes the primitives ADD and MUL which give the sum and product, respectively, of two atoms which are non-negative integers.

By allowing the operations of composition and recursion, the class of functions definable in terms of the primitive functions can be expanded to the set of partial recursive functions over the domain of S-expressions. McCarthy [11] gives a formal development of the class of functions computable in terms of given base functions. Any function belonging to the class of computable LISP functions can be described with the use of LAMBDA and LABEL notations and conditional forms.

These notations will be explained in terms of the syntax of Appendix A. Non-terminal symbols are in lower case letters. The LABEL notation looks like

@<FN; function>,

where FN is the name assigned to the function. The LABEL notation allows the programmer to define recursive functions. The LAMBDA notation looks like

&<<X;...;XN>; form>,

where X1 through XN are dummy variables used within the form which defines the function. When a LAMBDA function is applied to a set of S-expressions, the dummy variables are assigned the values of the corresponding S-expressions.

There are four possibilities for a form. It may be merely a constant or a variable. Or it may be another function with its own argument list. Or lastly, it may be a conditional form. In the syntax of Appendix A, conditional forms appear as a list of predicate-expression pairs. Conditional forms are evaluated by evaluating successive predicates until one of them evaluates to T. The value of the corresponding expression then becomes the value of the entire conditional form.

A more general notation for a conditional form is (p → c,a), where p is the premise, c is the conclusion, and a is the alternative. The premise is a propositional form which evaluates to a truth value. The value of the premise determines whether the conclusion or the alternative will give the value of the form. The conclusion and the alternative may themselves be conditional forms. Reference 11 includes a detailed discussion of the formal properties of conditional forms.

B. FUNCTIONALS AND PROGRAM ORGANIZATION

As defined by the syntax in Appendix A, an argument can be a form or a function. Functions which accept functions as arguments are called functionals. The best known functional in LISP is evalquote. Evalquote takes as arguments any LISP function and its argument list and gives the result of the function applied to its arguments, i. e.,

EVALQUOTE<function; (argument...argument)>

is equivalent to

function<argument;...;argument>.

Reference 10 describes evalquote. Appendix D contains a program listing of evalquote in the syntax of Appendix A.

The concept of functionals has powerful implications. One of the features of functionals is that they enable the programmer to achieve the same economy of expression and memory space that is achieved in other programming languages through the use of subroutines. A function can be defined once as an argument to a LAMBDA function. The function definition is then paired with a variable and can be used repeatedly in the defining form of the LAMBDA function.

The sample programs presented in this thesis make use of functionals in this way. The general organization of these programs is given below. Comments are enclosed in single quotes. Non-terminal symbols are in lower case letters.

```
&<<VAR1;...;VARN>;          'program variables'
     &<<F1;...;FM>;         'function names'
          form>             'defining form for the program
                             in terms of V1 through VN
                             and F1 through FM'
     <'F1' function;        'function definitions'
      'F2' function;
           ...
      'FM' function>>       'end of program function'
<s-exp1;...;s-expN>         'program argument list'
#                           'EOF symbol'
```

C. DATA FLOW GRAPH

A data flow graph is a graphical description of the execution of a specific program. The entities depicted by a data flow graph are data elements and primitive processes.

For the sample data flow graphs of this section, the nodes are labeled with primitive processes and the edges are

labeled with data elements. Figure 1 is the data flow graph for the FORTRAN function SUM (from Section I) where the data is an integer vector of eight 1's. Only the primitive '+' is used in the graph. Figure 2 is the data flow graph for the LISP function SUM (also from Section I) applied to a list of eight 1's. Again only the primitive 'ADD' is used in the graph. For a specific program execution the data flow graph illustrates the execution order among the primitive processes, specifically describing which processcs can be performed simultaneously and which must be performed in sequence.

In graph theory, an edge is usually defined by the two nodes to which it is connected. There are edges in the graphs of figures 1 and 2 which are connected to only one node. This is only a superficial discrepency which can be corrected by viewing the data elements as nodes and the primitive processes as edges. Figure 3 shows this dual form for the graph of figure 2. In the dual form, a binary primitive process is represented by two edges.

In graph theory a path is defined as any sequence of edges in which each successive edge originates from the terminal node of the preceding edge. A data flow graph is an acyclic directed graph. The term "directed" means that a direction is associated with each edge. "Acyclic" implies that no edges are repeated in any path. The length of a path is defined as the number of edges in the path. The length of the longest path in the dual form of a data flow graph represents the number of execution stages which would be required in a parallel execution.

The width of a data flow graph at a particular stage of execution is defined herein as the number of primitive processes to be executed at that stage. Hence, the maximum width represents the minimum number of processors required

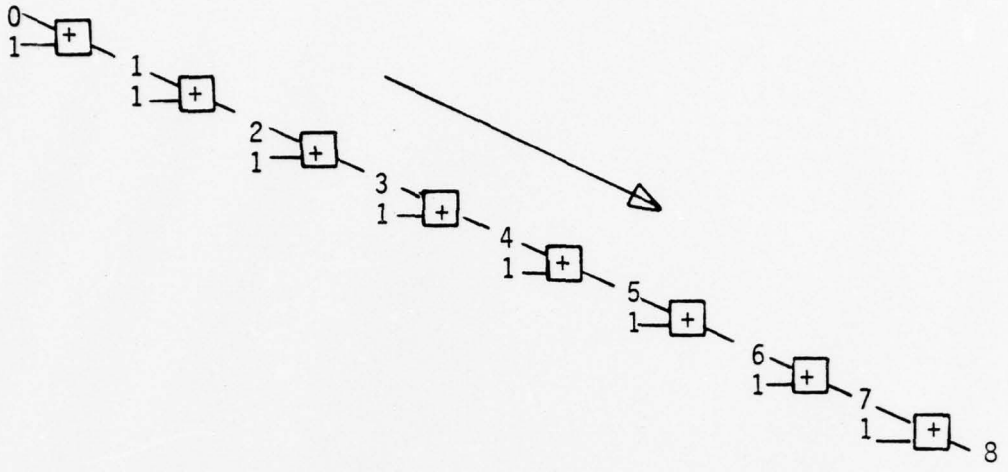


Figure 1. Graph of Fortran Sum

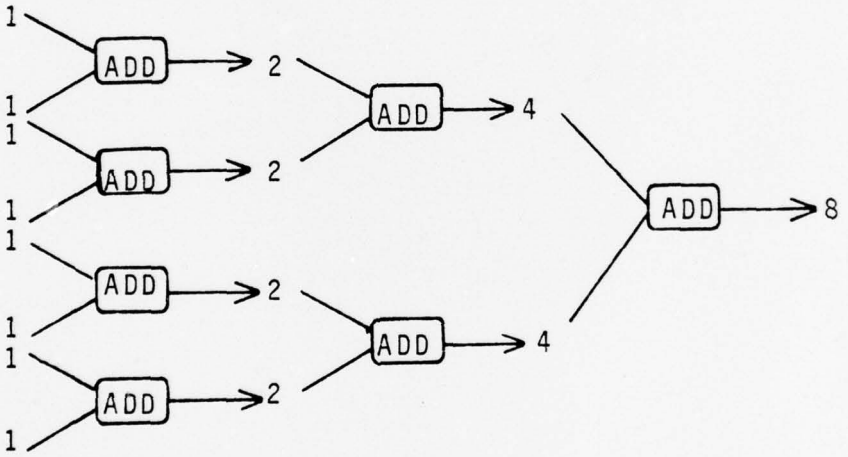


Figure 2. Graph of LISP Sum

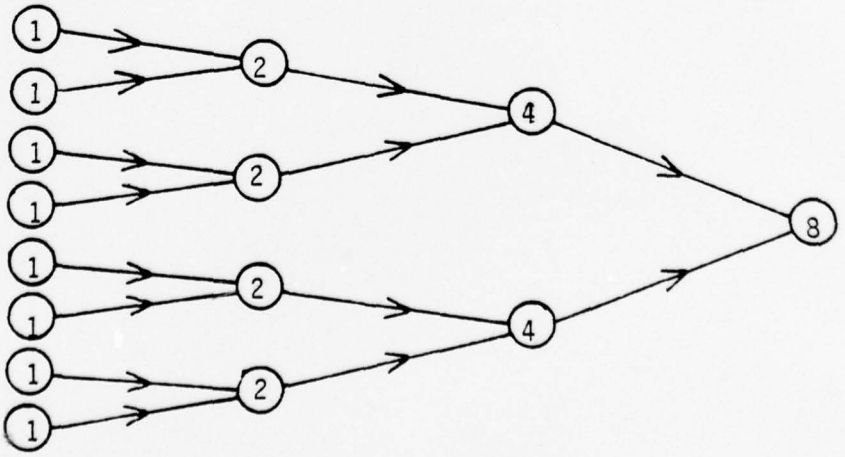


Figure 3. Dual Form of LISP Sum

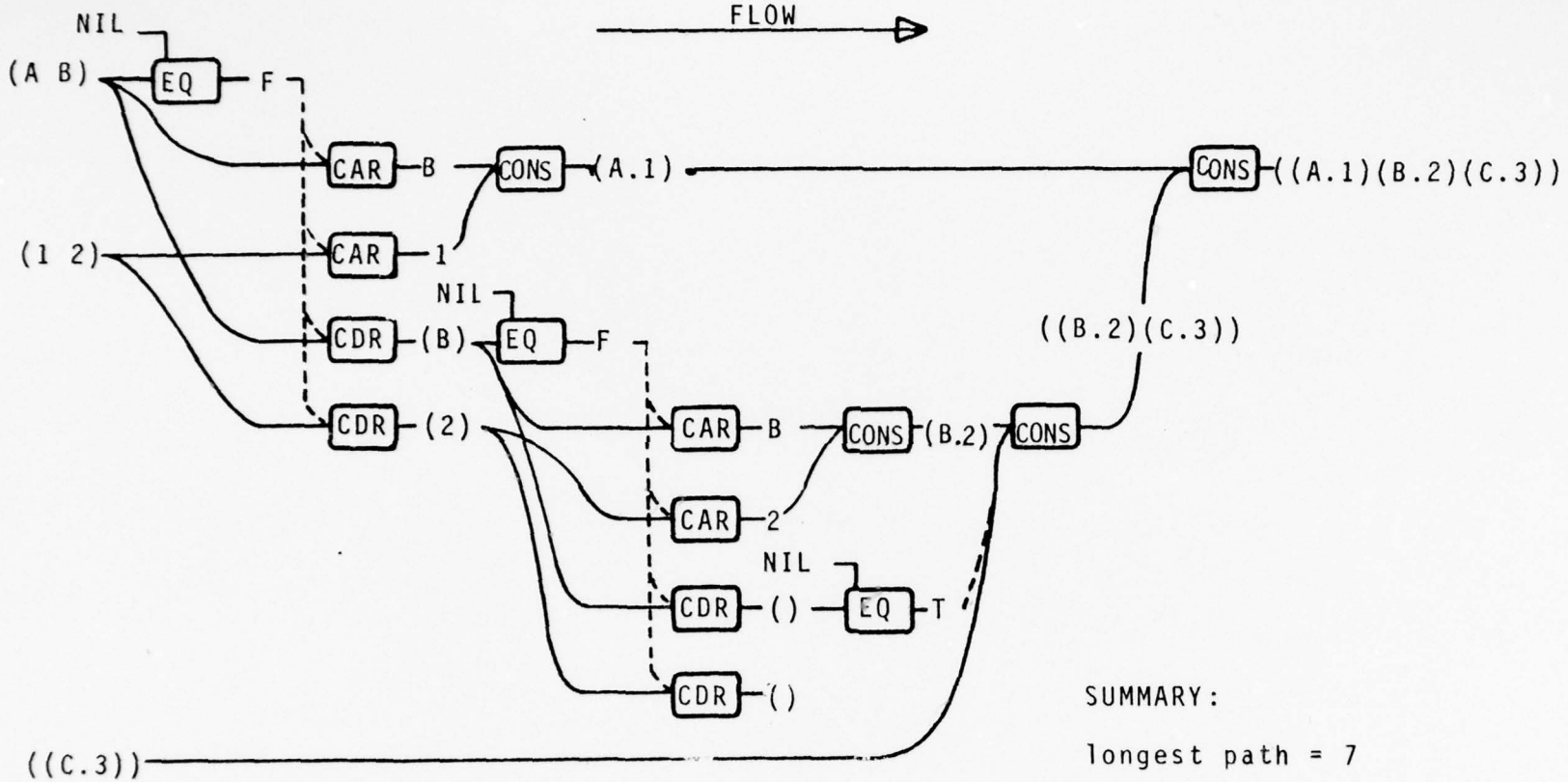
for optimum parallel execution.

Data flow graphs should not be confused with program graphs. Prcgram graphs represent abstractions of flow charts and are useful in the analysis of algorithms. They highlight the flow of control without regard to a particular data set. Program graphs are usually cyclic digraphs.

PAIRLIS is a LISP function which pairs together elements of two S-expressions and appends the resulting list of pairs to an existing list. PAIRLIS is used by evalquote to pair variables with their corresponding S-expression values and store these pairs on the association list. Program 1 of Appendix D is PAIRLIS<(A B);(1 2);((C.3))>. Figure 4 shows the data flow graph depicting the execution of this program. This graph includes all the primitives used in PAIRLIS and illustrates the order in which they are executed. Those primitives which line up vertically may be executed in parallel. Otherwise, the primitives are executed in order from left to right.

D. G-VECTOR

A g-vector (graph vector) is a list of integers which describes a data flow graph which in turn describes a program execution. The general form of a g-vector is $(w_1\ w_2\ \ldots\ w_n)$. The number of elements in the vector, n, represents the maximum path length of the data flow graph which is the minimum number of steps (or stages) required for the entire computation. Each element, w_i, of the g-vector represents the width of the data flow graph (or the



SUMMARY:

longest path = 7

maximum width = 4

total nodes = 15

speed-up ratio = 15/7

Figure 4. Data Flow Graph for PAIRLIS⟨(A B);(1 2);((C.3))⟩

minimum number of machine level processors required for optimum paralleling) at step i in the computation. The sum of the elements in the g-vector represents the total number of primitive processes performed in the computation.

Each data element in the data flow graph describing the execution of a LISP program has associated with it a g-vector. The g-vector describes a sub-graph that represents the computations performed to produce the data element. The g-vector may be empty as is the case with constants.

There are two binary operations which can be performed on a pair of g-vectors. A g-vector may be appended to another g-vector to produce a longer resultant g-vector. The resultant g-vector represents the data flow graph for two processes which were performed in sequence. A g-vector may be combined with another g-vector by summing their corresponding elements. The resultant "wider" g-vector represents the data flow graph for two processes which were performed in parallel. The g-vector for the data flow graph of figure 4 is (1 4 2 4 2 1 1).

III. EVALQUOTE2

Evalquote2 is a LISP function similar to Evalquote. For input, evalquote2 takes two S-expressions. The first S-expression represents any LISP function, and the second S-expression represents a valid argument list for that function. As with all LISP programs, the output from evalquote2 is a single S-expression. The CAR of this S-expression represents the result of the input function applied to the input argument list. The CDR of the output S-expression is the g-vector (a list of integers) describing the data flow graph resulting from the application of the input function to the input argument list. Appendix B is a listing of evalquote2 applied to the PAIRLIS function of Figure 4.

The sub-functions used to define evalquote2 are similar to the sub-functions used to define evalquote along with some additional functions used to compute the g-vector. These sub-functions will be discussed shortly, but first will be a discussion of the logic used by evalquote2 to compute the g-vector.

A. LOGICAL DEVELOPMENT

The arguments for a function are independent of one another and may be evaluated simultaneously (in parallel). The evaluation of each argument produces a resultant S-expression and g-vector. When an argument is a function with its own argument list, the g-vector associated with the

resultant data element describes the data flow graph determined by the application of the function to its argument list.

When all of the arguments have been evaluated, their associated g-vectors are combined to produce a single g-vector. The g-vectors are combined by summing their corresponding elements. The resultant g-vector describes the data flow graph which describes the parallel evaluation of the arguments. The length of the resultant g-vector will be equal to the length of the longest g-vector created in the evaluation of the argument list. Each element of the resultant g-vector represents the number of primitive functions that were executed at that stage in the parallel evaluation of the argument list.

When all the arguments have been evaluated, the function can be applied. The application of the function to the evaluated argument list is described by a new data flow graph. The g-vector for this graph is appended to the g-vector for the combined argument list to produce a longer g-vector. This longer g-vector describes the total data flow graph which represents the evaluation of both the function and its argument list.

When the defining form for a function is a conditional, the g-vector must be computed in a way which describes the evaluation of a conditional form. As discussed previously, the general form for a conditional is $(p \rightarrow c,a)$. The possibility of parallel evaluation of p, c, and a will be discussed later. Normally p (the predicate) is evaluated first and then c (the conclusion) or a (the alternative) is evaluated next depending on the value of p. Hence, the g-vector for p is computed first, and then the g-vector for c or a is appended. The resultant g-vector describes the sequential evaluation that has occurred.

B. SUB-FUNCTIONS

In order to more easily understand the following explanations of the sub-functions used in evalquote2, it may be helpful to scan Appendix B before proceeding.

Apply2 computes the g-vector describing the application of a function to its arguments. The parameters for apply2 are similar to those for apply except that the second parameter represents both the argument list and the combined g-vector describing the parallel evaluation of the arguments. Notice that when apply2 is first called by evalquote2, the g-vector for the argument list is empty. This is because the initial arguments are all S-expressions and need no evaluation. If the function is a primitive, then the g-vector describing the application is (1). Therefore, (1) is appended to the existing g-vector and this new g-vector is associated with the resultant data element.

If the function is a lambda or label expression, or a previously defined function, then the defining form will be evaluated by eval2. Eval2 will return a data element associated with a g-vector describing the evaluation. The sub-function compose is then used to append the g-vector returned by eval2 with the g-vector that came with the argument list.

Eval2 is similar to eval in that it evaluates forms. The difference is that eval2 associates a g-vector with an s-expression (resultant data element) for each evaluation. If the form is a variable or a constant the g-vector is empty. If the form is a conditional then evcon2 is called.

Evcon2 is similar to evcon except that it also returns a g-vector describing the evaluation of the conditional. Evcon2 evaluates the first predicate and calls on graphcon. If the predicate is true, graphcon evaluates the corresponding expression and calls compose. Compose appends the g-vector for the expression to the g-vector for the predicate and associates the resultant g-vector with the resultant data element. If the predicate is false, graphcon calls compose with the result of evcon2 applied to the remainder of the conditional and the g-vector of the first predicate.

If the form given to eval2 is a function with its argument list, the argument list is given to evlis2 for evaluation. Evlis2 is similar to evlis in that it evaluates arguments, but it also combines the g-vectors of evaluated arguments to produce a resultant g-vector describing the parallel evaluation of the arguments.

The sub-function compose is used to compute g-vectors resulting from the composition of functions. Composition of functions describes computational processes which must naturally occur in sequence. Compose is called from apply2 and graphcon. Append is a standard LISP function used to create a new list of the top level elements of two input lists. Append is called from apply2 and compose. Combine is used to compute g-vectors representing the parallel evaluation of arguments. Combine is called from evlis2. Sum is used by combine for adding corresponding elements of two g-vectors. The remaining sub-functions are identical to sub-functions used in evalquote.

IV. RESULTS OF EVALQUOTE2 IMPLEMENTATION

Evalquote2 has been implemented through an interpreter program written in Algol-W. This section documents the interpreter which has been compiled to run on a S/360. Also discussed are the sample LISP programs which were run under the interpreter and their results.

A. THE ALGOL-W INTERPRETER

Appendix E contains a source listing for the Interpreter. Functionally, the interpreter is nearly identical to evalquote2. That is, it produces the result of a LISP function applied to its arguments along with the associated g-vector. The following paragraphs summarize the organization of the Interpreter.

1. Input

For input, the Interpreter accepts programs written in the metalanguage syntax of Appendix A. Input is expected from 80-character records and can be written free-form (column independent).

If the first character of a record is a '$', the second character represents a toggle and causes a logical variable to be reset. The remainder of the record is ignored and may be used to comment on the reason for the toggle. If the toggle is a '$', it causes the toggle

records to be listed until the next '$$' record is encountered. '$L' resets the LISTING variable which is turned on initially. '$T' resets the TRANS variable which is turned off initially. When on, TRANS causes the S-expression translations of the input function and argument list to be printed on the output device. '$A' causes only the arithmetic operators (ADD and MUL) to be included in the computation of the g-vector.

If the first character of a record is a '*', the entire record is considered a comment. Single quotes are used to delimit in-line comments.

2. Translation

The SCANNER routine reads tokens (identifiers, constants, numbers, and specials) from the input stream. The translation routines change the program into two S-expressions (one for the function and one for the argument list) and store them in memory in the form of linked lists. The translation routines function in accordance with the translation rules of Appendix C. These translation rules were derived from the rules for translating M-expressions to S-expressions presented in Ref. 10.

3. Interpretation

Because Algol-W supports recursion, the interpretation routines are nearly identical to evalquote2 which was explained in the previous section. The evalquote2 procedure within the Interpreter may be viewed as a microprogram in a hypothetical LISP machine. The machine's memory already contains an S-expression for a function and an S-expression for an argument list. The machine generates

a resultant S-expression containing the program result and the g-vector.

4. Output

The output from the Interpreter includes the resultant S-expression containing the program result and g-vector and also a summary of the information contained in the g-vector. The summary includes the number of processing elements required for a parallel execution, the number of execution steps required for both a sequential and a parallel execution, and the speed-up ratio. Additional output from the Interpreter includes diagnostic error messages for the more common syntactic and semantic errors.

B. SAMPLE PROGRAMS

Appendix D contains the sample LISP programs which were run under the interpreter. Program 1 is the PAIRLIS function with the same arguments used to generate the data flow graph of Figure 4. Program 1 is included to illustrate the output from the Interpreter. The output includes the program listing followed by the S-expression translations of the function and argument list (enabled by the $T toggle). The CDR of the resultant S-expression can be compared with the data flow graph of Figure 4.

1. Matrix Multiplication

Programs 2 through 5 represent matrix multiplication of two 4 x 4 matrices in which all the elements are 1's. Hence, the resulting product matrix is a 4 x 4 of all 4's.

The S-expression representation for the first factor is in row major order while the second factor is in column major order.

Programs 2 and 3 use the same algorithm to compute the matrix product. Program 3 uses the $A toggle to include only arithmetic operations in the computation of the g-vector. The MATMUL function computes the rows of the product matrix by calling the row function. The row function computes the elements of each row by calling the dot function. The dot function computes the dot product of each row of the first factor with each column of the second factor. The dot function is defined so as to sequentially add the integer products of vector elements. As discussed in Section I, this is not the optimum way to define an associative process for a parallel processor.

Programs 4 and 5 use the sum function presented in Section I to optimize the summation required for each dot product. The g-vector computed for program 5 considers arithmetic operations only. Note that the results for program 5 correspond to the theoretic results discussed in Section I. That is, the number of required sequential steps is $2n^3-n^2$, or 112 for n=4. The number of parallel steps is $1 + \lceil \log_2 n \rceil$ or 3 for n=4. Because program 3 performs additions sequentially, it requires one more parallel step than program 5.

The speed-up ratios computed for programs 4 and 5 might be considered as lower and upper bounds, respectively, for an actual speed-up ratio (one that compares an actual parallel machine with an actual sequential machine). Program 5 ignores the data accesses represented by CAR, CDR,

and CONS operations and also execution controls represented by the EQ operation. Obviously, the data must be moved into position in order to be operated upon, even if the movement of data takes place in parallel. Program 4 computes the speed-up ratio by giving the same weight to CAR, CDR, CONS, and EQ as it gives to ADD and MUL. This is not necessarily a correct assumption either, since data can normally be accessed from a high-speed memory faster than the arithmetic operations can be performed.

2. Symbolic Differentiation

Programs 6 and 7 represent the symbolic differentiation of a fourth degree binomial, $(x + y)^4$, with respect to x. The function consists of two primary sub-functions. DIFF computes the derivative by the rules for differentiating algebraic expressions. SIMP simplifies the result by eliminating factors of "1" and addends of "0."

For program 6 the S-expression representation of $(x + y)^4$ is

(((x + y) * (x + y)) * ((x + y) * (x + y))).

In this expression the data is arranged symmetrically. For program 7 the data is arranged asymmetrically and looks like

((x + y) * ((x +y) * ((x + y) * (x + y)))).

As expected the speed-up ratio is greater for program 6 (5.2) than for program 7 (4.1). This comparison was made to provide an example in which symmetrically organized data caused a greater speed-up ratio than the same data organized asymmetrically.

3. Evalquote

Program 8 is the universal function evalquote. The arguments for evalquote are the S-expression translations from program 1. The speed-up ratio for program 8 is 1.86. This is the smallest speed-up ratio of all the sample programs. Additional runs were made with this program in which the PAIRLIS function paired lists of three elements each and lists of four elements each. Each run produced essentially the same speed-up ratio (1.86).

For the matrix multiplication examples, the data is two-dimensional. As the size of the matrix factors is increased, the resulting data flow graph widens at a greater rate than it lengthens. In fact it widens at a rate proportional to n^3 (the number of simultaneous multiplications) while it lengthens at a rate proportional to $\log_2 n$. Program 8, on the other hand, is operating on one-dimensional data. Increasing the size of the data elements for PAIRLIS causes the total number of elements in the data flow graph to increase at the same rate as the length increases. Hence, the speed-up ratio remains essentially constant.

V. HARDWARE CONSIDERATIONS

The problem addressed in this section is how to design a hardware system which will implement the parallel execution of algorithms which are defined by a software system such as "pure" LISP. A detailed hardware design will not be given. Rather, a "skeletal" design for the hardware will be presented at a functional level in order to bring out some of the considerations involved in any design.

Before proceeding with the example design, some clarification of terminology will be given. A parallel processing system refers to both the software and hardware portions of a complete system. A parallel processing machine (or computer) refers to the hardware alone, including at least memory and processors. A parallel processing system might include one or more parallel processing machines. The processor module refers to the module within a machine which contains the processing elements. A processing element refers to a single processor.

A parallel processing machine must perform a function similar to evalquote2. That is, it must evaluate a LISP function applied to its arguments and in the process recognize parallelism. The data for this machine, both functions and operands, is in the form of ordered sets (parenthetical expressions). This data can be stored in a sequentially organized memory in the form of linked lists.

A. PROCESSOR MODULE

Ideally, the processor module would be constructed on a single chip. Figure 5 is a modular diagram of a processor module. Each processing element in the module represents a hardware implementation of evalquote. There are three inputs to a processing element. The first input is a memory address for a program or a form as defined by the metalanguage syntax. The second input is a device address for returning the result to be computed. The third input is the memory address of the applicable association list. The outputs from a processing element are the result of the function applied to its arguments and the address of the device for which this result is destined.

1. The Processor Manager

The processor manager controls data transfers between the processing elements. The processor manager also keeps track of the status (busy or free) of each processing element. Initially, a LISP program (a list containing a function and constant arguments) is made available to a processor module from an external agent such as a terminal user or another parallel processing machine. After the program is read into memory, the processor manager assigns the program to a processing element along with a return address to an external device (terminal, printer, external storage, or another parallel processing machine). As the processing element recognizes processes which can be computed in parallel these processes are made available to the processor manager for assignment to other available processing elements. The return address for these processes will be the processing element that originated them. When all processing elements are busy the processor manager will queue waiting processes.

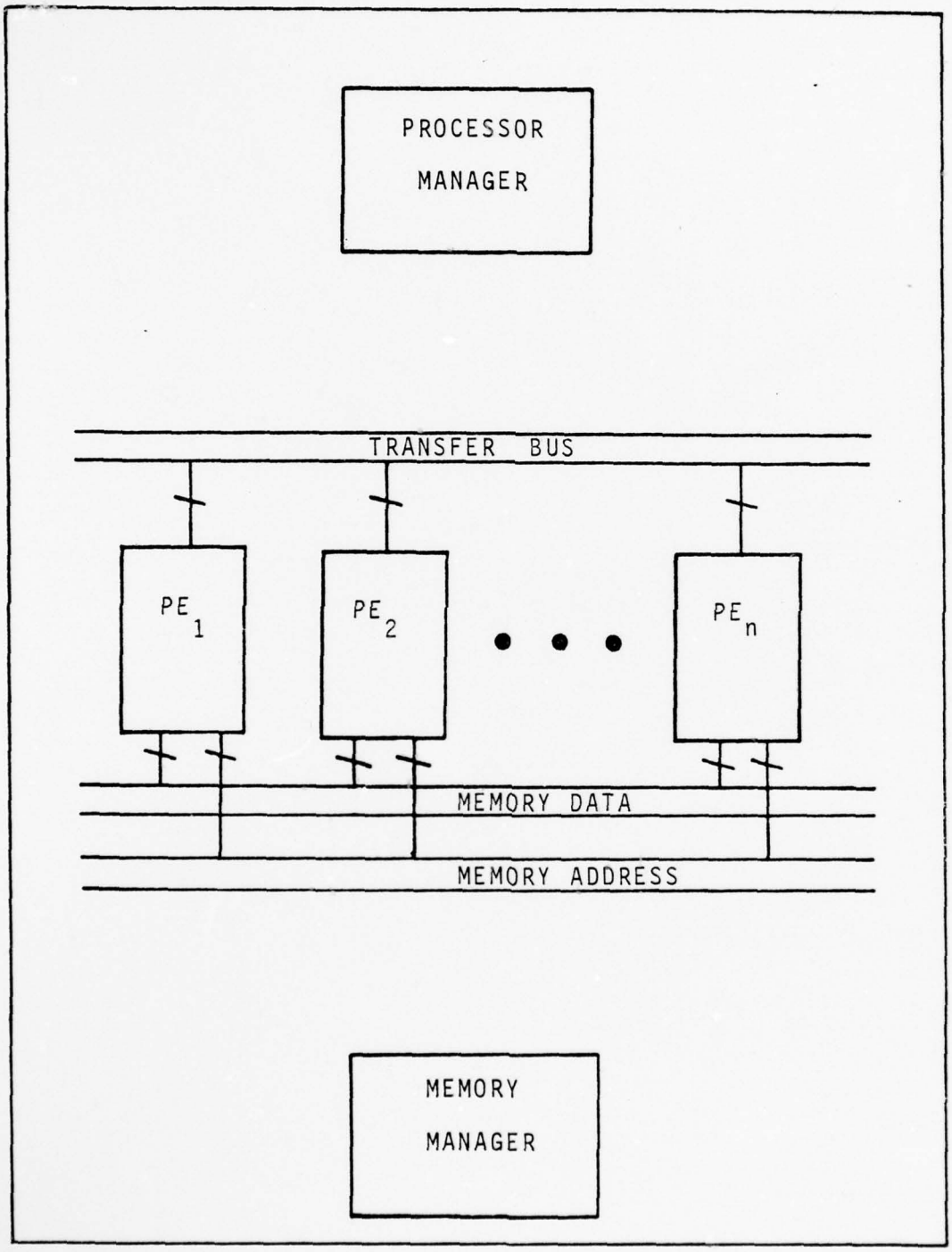


Figure 5. Processor Module

2. The Memory Manager

The memory manager controls the common memory. Between programs, the memory manager converts memory into a single list of free storage cells. This list is made available for reading a new program into memory. Once execution has begun, the memory manager provides free storage cells to each processing element for use in performing the CONS primitive. Garbage collection is performed by monitoring the association-list stack in each processing element and returning links from outdated lists to the free storage list.

3. Timing

By constructing the processor module on a single chip it can be controlled by a single timer. Two basic clock cycles, a compute cycle and a transfer cycle, are required.

The compute cycle enables all the processing elements to perform a computation if they have one to perform. Also during this cycle, the processor manager's list of available processors is updated via the status flags on each of the processing elements. During the compute cycle the memory manager can perform garbage collection or issue free storage cells.

During the transfer cycle the processor manager performs a linear sweep of the processing element output ports. As an output port with data to be transferred is swept, the destination (indicated by the return address) is enabled and the data transferred. One entire sweep is

performed in a single transfer cycle. Simultaneously, during the transfer cycle, the memory manager performs a linear sweep of the memory ports for each processing element. All memory read and write operations are performed during a single transfer cycle.

B. A PROCESSING ELEMENT

Figure 6 is a modular diagram of a single processing element. The purpose of a processing element is to accept, as input, a form, as defined by the metalanguage systax, and to produce, as output, the evaluated result of the form. The actions of the processing element are controlled by the decode-and-control module (DCM). The DCM contains the microprograms for all the primitive functions (CAR, CDR, CONS, ADD, etc.). The DCM also manages four pushdown stacks which are required for evaluating complex forms. The DCM also controls the processing element status register and the I/O to the processor transfer bus and the memory bus. Figure 7 is a functional flow chart describing the tasks performed by the DCM.

The inputs to a processing element are provided by the processor manager or by the processing element's own DCM. The first input is the address of a form and goes into the program register. The second input is the address of the association list for the form. If the form is an original program (function and constant arguments) from an external device, then the association list will be empty. The incoming return address register receives the address of the device to which the result will be sent.

C. DECODE AND CONTROL MODULE

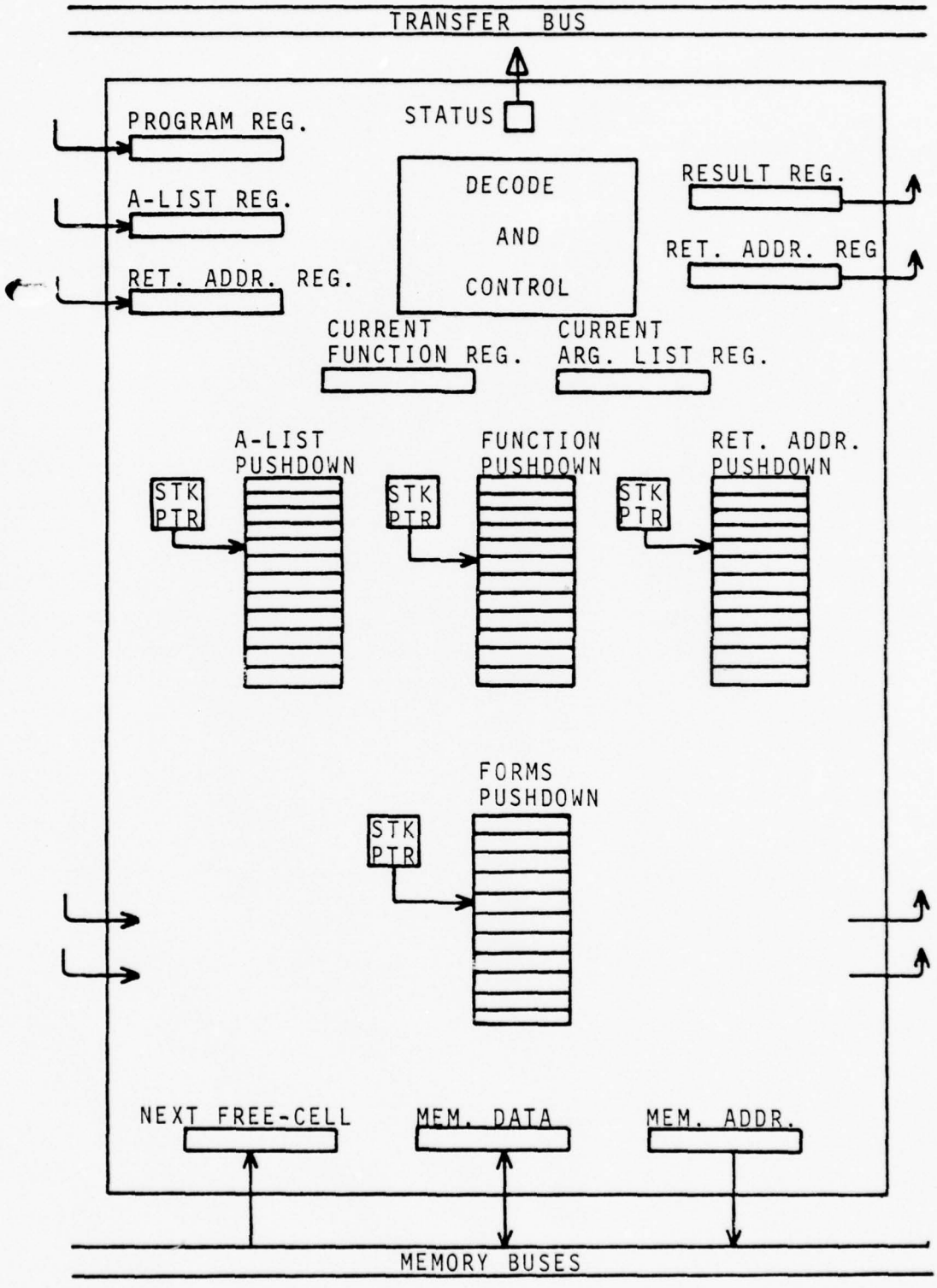


Figure 6. Processing Element

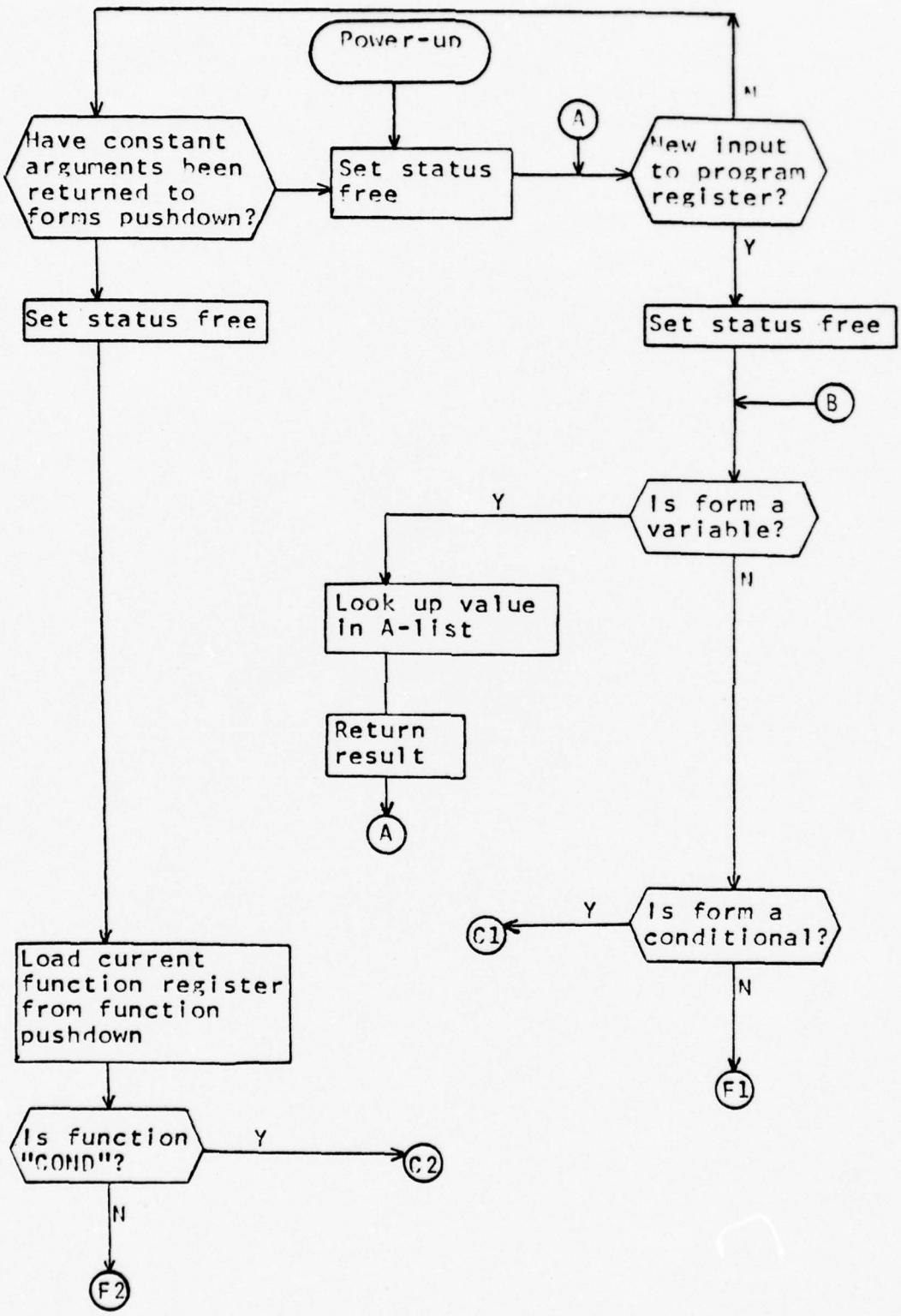


Figure 7. Flow Chart for Decode and Control Module (1 of 3)

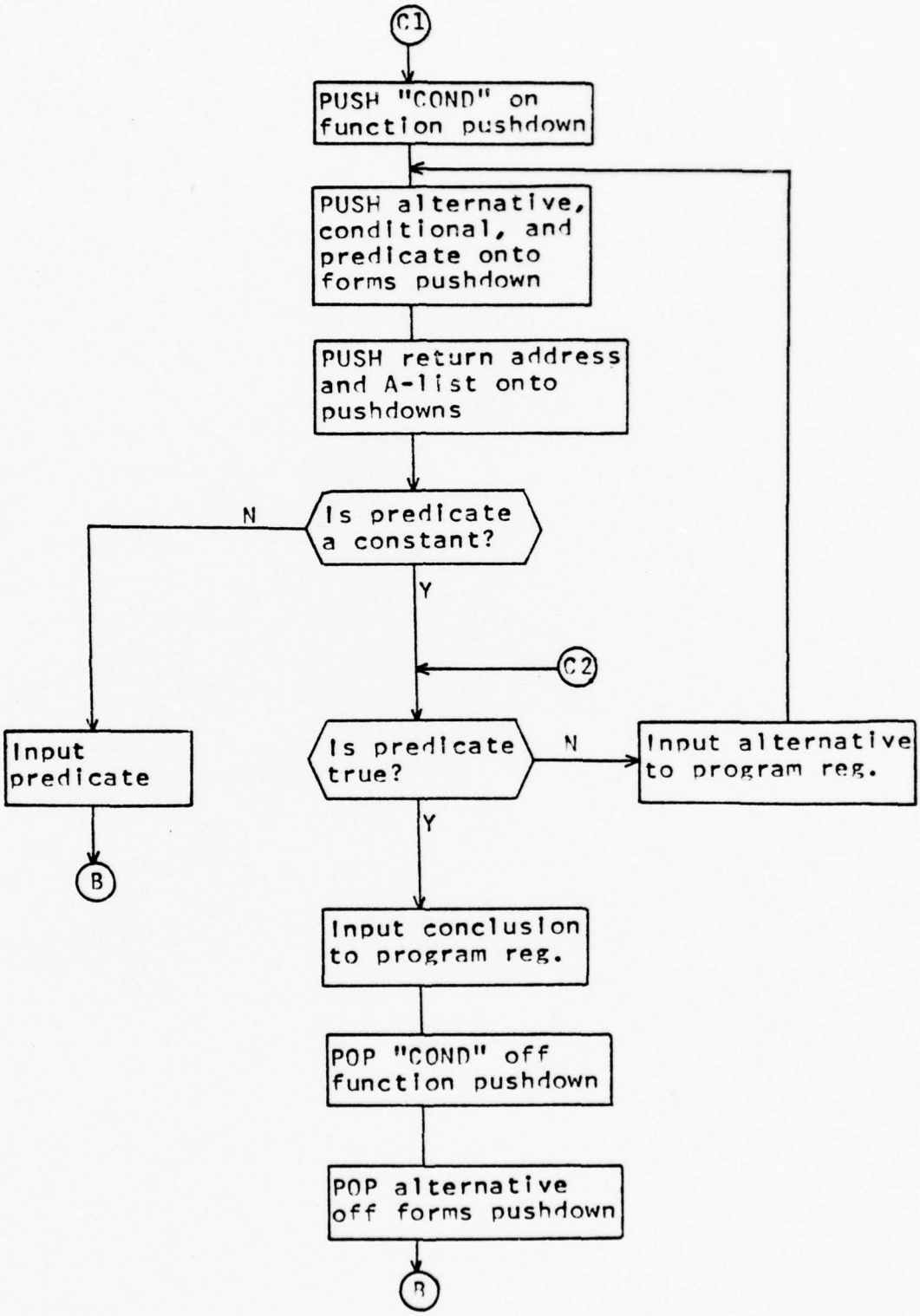


Figure 7. Flow Chart for DCM (2 of 3)

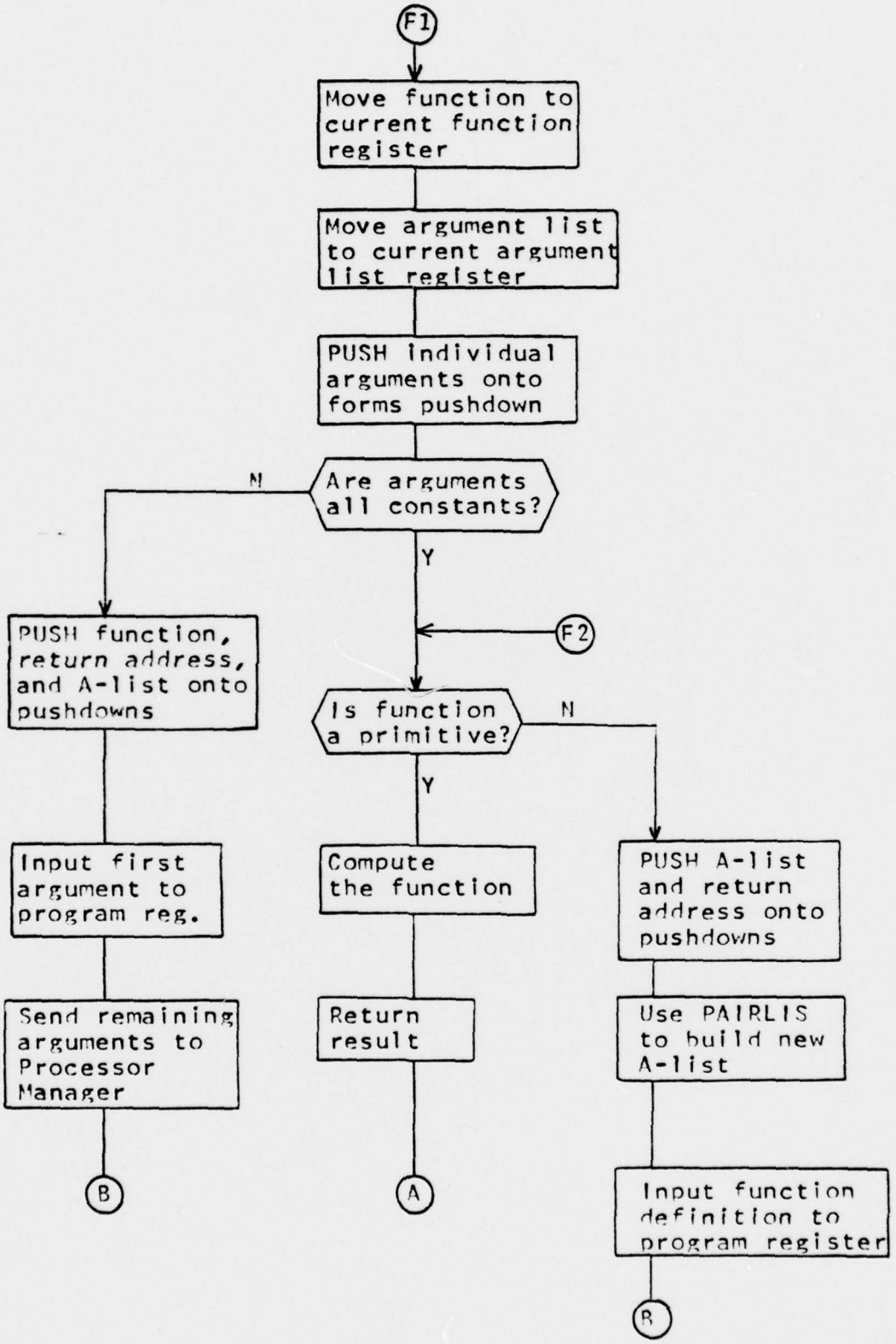


Figure 7. Flow Chart of DCM (3 of 3)

The DCM is designed to implement the overall policy of the parallel processing system. That policy is to perform sequential processes in sequence and to enable parallel processes to be performed in parallel. Hence, if the form that is input to the program register is a function and a set of unevaluated arguments, then the DCM will stack the function on the function pushdown, input one of the arguments to its own processing element, and send the remaining arguments to the processor manager for parallel evaluation by other available processing elements. When all the arguments have been evaluated and returned to the forms pushdown, the DCM will pop the function into the current function register. If the function is a primitive, it is decoded by the DCM and applied to the arguments. If the function is a lambda expression, the DCM creates a new association list in memory by pairing the variables of the lambda expression with the evaluated arguments on the pushdown. The DCM then inputs to the program register the remainder of the lambda expression which is a form defining the function.

If the form is a conditional, the reserved word COND is stored on the function pushdown. The list address for the second and successive predicate-expression pairs, the list address for the first expression, and the list address for the first predicate are stored in that order on the forms pushdown. Those three forms represent the predicate, the conclusion, and the alternative, respectively, for a generalized conditional expression. it is input to the processing element for evaluation. If the predicate is not a constant, If the predicate evaluates to true, the conclusion, which is next on the forms pushdown, is evaluated, the alternative is discarded, and the COND is popped off the function stack. If the predicate evaluates to false, the COND is left on the function stack, the

conclusion is discarded, and the alternative conditional expression is input to the processing element for evaluation.

When the form input to a processing element is a variable, the DCM uses the accompanying association list to search for the corresponding constant value.

VI. ADDITIONAL PARALLEL PROCESSING CONSIDERATIONS

There remain several areas within the realm of parallel processing of recursive functions which need further research. Three of these areas will be discussed in this section.

A. PARALLEL PROCESSING OF CONDITIONAL FORMS

Parallel processing of a conditional form means that evaluations of the predicate, conclusion, and alternative are begun in parallel. If any of these forms are themselves conditionals, or contain conditionals, then they too are processed in parallel. When the predicate evaluates to true (or false), all processing generated by the alternative (or conclusion) is halted, and any storage allocated for the evaluation of the alternative (or conclusion) is reclaimed.

One of the problems with parallel processing of conditional forms is that it is wasteful of memory. It might happen that parallel processing of a conditional form would exhaust memory before completion, whereas normal processing could complete within available memory. Problems associated with limited memory sizes, however, can be expected to lessen as advances in memory technologies continue to push cost down and volume up.

Another problem with parallel processing of conditional forms concerns undefined forms. A conditional form is considered defined if: 1) the predicate is defined; and 2)

the conclusion is defined if the predicate is true, or the alternative is defined if the predicate is false. Hence, a well-defined conditional form may have either an undefined conclusion or an undefined alternative, but not both. A system which processes conditional forms in parallel must be prepared to deal with undefined forms. Some undefined forms are recognizable while others are not. For example, if X is an atom, then CAR<X> can be recognized as undefined. Analysis of the halting problem has shown that some undefined forms (e.g., some which recur infinitely) may not be recognizable.

Assuming that a parallel processing system has sufficient resources (memory and processors), it may still be possible to process conditional forms in parallel. For example, assume the alternative is undefined. As soon as the predicate evaluates to true, processing could be stopped on the alternative. This may be a difficult and time-consuming task, however, since the alternative process may have tied up an intricate net of processing elements.

Because there are so many unanswered questions concerning parallel processing of conditional forms, evalquote2 was designed to graph conditionals in the traditional way.

B. THE MEANING OF SPEED-UP RATIOS

In the dual form of the data flow graph (the one where the nodes represent data elements and the edges represent primitive operations), the speed-up ratio can be defined as the number of nodes with indegree greater than zero divided by the maximum path length. This definition presupposes an unweighted graph which is probably not true for any actual

implementation. For example, the procedures for ADD and MUL in the Interpreter are much more complex than the procedures for CAR and CDR. This was the primary reason for the Interpreter's $A toggle which causes only the arithmetic primitives to be graphed. It is not inconceivable to imagine a parallel machine which has a "smart," associative memory that performs the CAR and CDR functions automatically, thus eliminating them from the data flow graph altogether.

In a machine where each instruction (primitive) requires several timer states to complete, the data flow through the primitive processes can be described by a weighted graph. The weight assigned to each primitive represents the number of timer states required for that primitive. There is still the problem of coordinating data transfers between processing elements which implies a need for synchronization. Each stage in the parallel execution could be timed to allow for the longest possible instruction. This would be analogous to the unweighted graph. Or each stage could be timed to the longest instruction in the stage. This could be implemented by each processing element setting a ready line at instruction completion. The processor manager would begin data transfers when all the ready lines were set. A third alternative is to let each processing element execute sequentially until there is data to be transferred (a completed rusult or arguments to be computed in parallel), and then to set a transfer ready line. The processor manager would continuously monitor the transfer ready lines. When a line goes true, the processor manager would interrupt the destination and enable the transfer.

C. THE WIDTH COMPUTATION

The width at each stage of a data flow graph has been defined rather loosely as the number of primitive processes to be executed at each stage. More precisely, the width at stage i has been computed as the number of distinct primitives representing step i in separate paths. The significance of the width of a data flow graph is that it represents the number of processors required at each stage for a program execution represented by that data flow graph. But what if the required number of processors for a given stage were not available? Is there a way for some of the processes to be delayed to future stages when sufficient processors are available and yet not increase the number of parallel steps required for the entire execution?

Consider once more the data flow graph of Figure 4. If only three processors were available at stage two, it might be possible to delay the first CAR operation, and hence the first CONS operation, and still complete the execution in seven parallel steps. Delaying the first CONS operation by one stage would cause the width of the graph at stage 4 to increase from 4 to 5. This increase could be avoided by further delaying this CONS operation one or two more stages.

If the first CDR operation at stage 2 were delayed, it would obviously cause an increase in the number of parallel stages required for completion. How can the proper process to be delayed be recognized? This would apparently require some "look-ahead" capability not included in the parallel system of the previous section. Without "looking ahead," it may be possible to adopt a strategy which causes the "right" processes to be delayed most of the time. For example, of the four processes at stage 2 in Figure 4, two produce arguments for a CONS and two produce arguments for the next invocation of the PAIRLIS function. The two CAR's and subsequent CONS represent a known number of required stages

(2), whereas the PAIRLIS function is recursive and the number of stages required will depend on the data.

Hence, one strategy might be to "tag" recursive functions, and when insufficient processors are available, to delay non-recursive functions before delaying recursive functions.

Another strategy that might be used to execute the program in a minimum number of stages with a limited number of processors requires a modification to the method of evaluation used by evalquote2 (and also evalquote). These universal functions evaluate another function and its arguments by first evaluating all the arguments and then applying the function. There are cases where some work can be done in applying the function before all of the arguments are evaluated. The PAIRLIS function of Figure 4 again serves as an example. From the data flow graph it can be seen that the second CDR of stage 2 (as well as the second CDR of stage 4) could be delayed one stage without affecting the total number of stages required. To do this would mean commencing the second (and third) invocations of PAIRLIS before all the arguments were evaluated. In other words, as soon as the first argument was evaluated, it would enable the first predicate, which only requires the first argument, to be executed.

The goal of the two strategies mentioned so far is to allow a reduced set of processors to still perform the program execution in a minimum number of stages. These two strategies as well as the goal they are seeking represent an area for further research in the design of a parallel processing system.

VII. SUMMARY AND CONCLUSIONS

It seems appropriate, in summary, to abstract from the preceding pages some organizational principles for the construction of a parallel processing system. These principles are inspired in part by the principles for recursive machines presented in Ref. 2. The three principles presented here represent the essential qualities of a system designed for the parallel processing of recursive functions. These principles are as follows.

1. Programming language operators (functions) are defined recursively in terms of machine-level operators.
2. Parallel tasks are distributed among available processors so that, at any time during program execution, the internal machine structure, i. e. the relationships between processors, represents the structure of the executing program.
3. Processors share a common main memory in which the data is stored associatively.

The first principle is quite similar to the first principle for recursive computer organization discussed in Ref. 2. By defining operators of the programming language as recursive functions composed of previously defined functions which are ultimately defined in terms of machine-level functions, there are no limits to the language levels possible. And yet, no matter how complex the language operators become, the programmer is still programming essentially in machine language, thus eliminating the need for intermediate compilation.

With such a language structure, a user could define a set of functions which would represent a special purpose programming language for his particular problem area, rather than having to adapt to a general purpose "high-level" language.

The second principle implies the need for some complex intercommunication scheme between processors working on the same problem. For example, should each processor be connected to all other processors, or should processors be arranged in some ideal network that provides "sufficient" intercommunication? The example parallel processing system proposed in the previous section suggests a single transfer bus controlled by a processor manager. This "conveyor" method has been included in earlier proposals for parallel systems [12].

The second principle also implies the concept of space-sharing as opposed to time-sharing. A user program from a peripheral terminal would be allocated available processors until completion rather than being paged in to a single processor for a time-slice. As the time-slice prevents a single user from monopolizing a time-sharing system, similar controls could be provided in a parallel system by limiting the number of processing elements or processing modules available to a single user program.

The third principle suggests sharing a main memory among the processing elements. This would eliminate the excessive data transmissions that would occur if each processor had its own memory. Storing the data associatively implies any scheme in which the data is arranged to facilitate accessing successive data elements. This principle has been implemented in the past (and in the Interpreter) by building linked-list data structures in linearly-organized memories.

For a memory which stores data in the form of S-expressions, the CAR and CDR functions applied to a data element represent access operations to the successor elements of that data element. These two functions could be performed automatically by a "smart" memory, and the successor elements always made available to a processing element if and when they are needed.

The concepts of parallel computation are not new. The literature is rich with proposals for parallel machines, summaries of such proposals, methods for recognizing parallelism, and other related subjects. Parallel machines of the past have been costly to construct and have required complex software support. Meanwhile, sequential machines have continued to achieve faster execution speeds. But now, as the increases in execution speeds begin to level-off, and LSI technology brings the cost of parallel systems within reason, the stage is set for a new and different generation of computing machines. A proposal has been given for a parallel processing system based on defining algorithms as recursive functions. Evalquote2 has demonstrated that the structure of algorithms defined as recursive functions makes possible the distinction between parallel and sequential tasks. The ideas presented in this thesis represent an attempt to show that a parallel system based on simple, highly-structured software can realize the speed-up of parallel computation while avoiding the burden of complex software.

APPENDIX A

SYNTAX

The non-terminal symbols are in lower case letters. The terminal symbols include <, >, ;, &, @, ¬, ., ", (,), upper case letters, and decimal digits. '::=' means "is defined as." '|' means "or." '...' means any number of the specified element.

```
program     ::= function<s-exp;...;s-exp>

function    ::= identifier |
                &<<variable;...;variable>; form> |
                @<identifier; function>

form        ::= constant | variable |
                function<argument;...;argument> |
                <form ¬ form;...;form ¬ form>

argument    ::= form | function

variable    ::= identifier

constant    ::= "atom" | (s-exp.s-exp) |
                (s-exp...s-exp)

s-exp       ::= atom | (s-exp.s-exp) |
                (s-exp...s-exp)

atom        ::= identifier | number

identifier  ::= letter | identifier letter |
                identifier digit
```

```
number      ::= digit | number digit

letter      ::= A | B | C | D | E | F | G | H | I |
                J | K | L | M | N | O | P | Q | R |
                S | T | U | V | W | X | Y | Z

 digit      ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

APPENDIX B

EVALQUOTE2

```
1     * EVALQUOTE2
2
3     &<<FN; ARGS>;
4
5         'SUB-FUNCTION NAMES'
6         &<<APPLY2; EVAL2; EVCON2; GRAPHCON; EVLIS2;
7            COMPOSE; COMBINE; APPEND; SUM;
8            PAIRLIS; ASSOC; NULL;
9            CAAR; CADR; CDAR; CADDR; CADAR>;
10
11            'DEFINITION OF EVALQUOTE'
12            APPLY2<FN; CONS<ARGS; ()>; "NIL">>
13
14        'SUB-FUNCTION DEFINITIONS'
15
16        <'APPLY2' &<<FN; X; A>;
17            <ATOM<FN> ¬
18               <EQ<FN; "CAR"> ¬
19                    CONS<CAAR<CAR<X>>; APPEND<CDR<X>; (1)>>;
20                EQ<FN; "CDR"> ¬
21                    CONS<CDAR<CAR<X>>; APPEND<CDR<X>; (1)>>;
22                EQ<FN; "CONS"> ¬
23                    CONS<CONS<CAAR<X>; CADAR<X>>;
24                        APPEND<CDR<X>; (1)>>;
25                EQ<FN; "ATOM"> ¬
26                    CONS<ATOM<CAAR<X>>; APPEND<CDR<X>; (1)>>;
27                EQ<FN; "EQ"> ¬
28                    CONS<EQ<CAAR<X>; CADAR<X>>;
29                        APPEND<CDR<X>; (1)>>;
30                "T" ¬ APPLY2<CAR<EVAL2<FN;A>>; X; A>>;
31            EQ<CAR<FN>; "LAMBDA"> ¬
32                CONPOSE<EVAL2<CADDR<FN>;
33                    PAIRLIS<CADR<FN>; CAR<X>; A>>; CDR<X>>;
34            EQ<CAR<FN>; "LABEL"> ¬
35                APPLY2<CADDR<FN>; X;
36                    CONS<CONS<CADR<FN>; CADDR<FN>>; A>>>>;
37
38        'EVAL2' &<<E; A>;
39            <ATOM<E> ¬ CONS<CDR<ASSOC<E; A>>; "NIL">;
40             ATOM<CAR<E>> ¬
41                <EQ<CAR<E>; "QUOTE"> ¬ CONS<CADR<E>; "NIL">;
42                 EQ<CAR<E>; "COND"> ¬ EVCON2<CDR<E>; A>;
43                 "T"¬ APPLY2<CAR<E>; EVLIS2<CDR<E>; A>; A>>;
44            "T" ¬ APPLY2<CAR<E>; EVLIS2<CDR<E>; A>; A>>>;
45
46        'EVCON2' &<<C; A>; GRAPHCON<EVAL2<CAAR<C>; A>; C; A>>;
47
48        'GRAPHCON' &<<P; C; A>;
49            <CAR<P> ¬ COMPOSE<EVAL2<CADAR<C>; A>; CDR<P>>;
50             "T" ¬ COMPOSE<EVCON2<CDR<C>; A>; CDR<P>>>>;
51
52        'EVLIS2' &<<L; A>;
53            <NULL<L> ¬ "NIL";
54             "T" ¬ COMBINE<EVAL2<CAR<L>; A>; EVLIS2<CDR<L>; A>>>>;
55
56        'COMPOSE' &<<X; Y>; CONS<CAR<X>; APPEND<Y; CDR<X>>>>;
57
58        'COMBINE'  &<<U; V>;
59            <NULL<V> ¬ CONS<CONS<CAR<U>; "NIL">; CDR<U>>;
60             "T" ¬ CONS<CONS<CAR<U>; CAR<V>>;
61                        SUM<CDR<U>; CDR<V>>>>>;
```

```
62
63          'APPEND' &<<X; Y>; <NULL<X> ¬ Y;
64              "T" ¬ CONS<CAR<X>; APPEND<CDR<X>; Y>>>>;
65
66          'SUM' &<<X; Y>;
67              <NULL<X> ¬ Y; NULL<Y> ¬ X;
68               "T" ¬ CONS<ADD<CAR<X>; CAR<Y>>;
69                         SUM<CDR<X>; CDR<Y>>>>>;
70
71          'PAIRLIS' &<<X; Y; A>;
72              <NULL<X> ¬ A; "T" ¬ CONS<CONS<CAR<X>; CAR<Y>>;
73                  PAIRLIS<CDR<X>; CDR<Y>; A>>>>;
74
75          'ASSOC' &<<X; A>;
76              <EQ<CAAR<A>; X> ¬ CAR<A>; "T" ¬ ASSOC<X; CDR<A>>>>;
77
78          'NULL' &<<L>; EQ<L; "NIL">>;
79
80          'CAAR' &<<L>; CAR<CAR<L>>>;
81
82          'CADR' &<<L>; CAR<CDR<L>>>;
83
84          'CDAR' &<<L>; CDR<CAR<L>>>;
85
86          'CADDR' &<<L>; CADR<CDR<L>>>;
87
88          'CADAR' &<<L>; CAR<CDAR<L>>>>
89
90
91      * SAMPLE ARGUMENTS FOR EVALQUOTE
92
93      <<LABEL PAIRLIS (LAMBDA (X Y A)
94          (COND ((EQ X (QUOTE NIL)) A)
95           ((QUOTE T) (CONS (CONS (CAR X) (CAR Y))
96                            (PAIRLIS (CDR X) (CDR Y) A))))));
97       ((A B) (1 2) ((C.3)))>
98      ##

****** EVALUATION BEGINS ******

RESULT IS:
(((A.1) (B.2) (C.3)) 1 4 2 4 2 1 1)


FREE STORAGE REMAINING:          2768
039.08 SECONDS IN EXECUTION
```

APPENDIX C

TRANSLATION RULES

Rules for translating programs:

1. A function is translated by the rules for translating functions.
2. <s-esp;...;s-exp> translates to (s-exp...s-exp).

Rules for translating functions:

3. &<<X;...;XN>; form> translates to (LAMBDA (X1...XN) form*) where form* is the translation of a form.
4. @<FN; function> translates to (LABEL FN function*) where function* is the translation of a function.
5. If a function is an argument, then it translates to (QUOTE function*).

Rules for translating forms:

6. "X" translates to (QUOTE X).
7. If the form is a parenthesized s-expression, then it translates to (QUOTE (s-exp)).
8. function<argument;...;argument> translates to (function* argument*...argument*), where argument* is the translation of an argument which can be a form or a function.
9. <form ¬ form;...;form ¬ form> translates to

(COND (form* form*)...(form* form*)).

APPENDIX D

SAMPLE PROGRAMS

Program 1.

```
$$
$T
   1     @<PAIRLIS; &<<X;Y;A>;
   2          <EQ<X;"NIL"> ¬ A;
   3           "T" ¬ CONS<CONS<CAR<X>;CAR<Y>>;
   4                     PAIRLIS<CDR<X>;CDR<Y>;A>>>>>
   5
   6     <(A B);(1 2);((C.3))>
   7     #


****** TRANSLATION FOLLOWS ******

(LABEL PAIRLIS (LAMBDA (X Y A) (COND ((EQ X (QUOTE NIL)) A) ((QUOTE T) (

CONS (CONS (CAR X) (CAR Y)) (PAIRLIS (CDR X) (CDR Y) A))))))

((A B) (1 2) ((C.3)))


****** EVALUATION BEGINS ******

RESULT IS:
(((A.1) (B.2) (C.3)) 1 4 2 4 2 1 1)


PROCESSORS REQUIRED FOR OPTIMUM PARALLELING.              4
EXECUTION STEPS (PARALLEL).....                           7
EXECUTION STEPS (SEQUENTIAL).....                        15
SPEED-UP RATIO (SEQUENTIAL/PARALLEL).....          2.142857
FREE STORAGE REMAINING:          15925
002.96 SECONDS IN EXECUTION
```

Program 2.

```
1     *       THIS FUNCTION PERFORMS MATRIX MULTIPLICATION.
2     *       DOT PRODUCTS ARE COMPUTED BY PERFORMING SEQUENTIAL
3     *       ADDITIONS OF INTEGER PRODUCTS.
4
5     &<<A;B>;
6         &<<MATMUL; ROW; DOT; NULL>;
7
8             MATMUL<A;B>>
9
10    <'MATMUL' &<<X;Y>; <NULL<X> ¬ "NIL";
11        "T" ¬ CONS<ROW<CAR<X>; Y>; MATMUL<CDR<X>; Y>>>>>;
12
13    'ROW' &<<R;C>; <NULL<C> ¬ "NIL";
14        "T" ¬ CONS<DOT<R; CAR<C>>; ROW<R; CDR<C>>>>>;
15
16    'DOT' &<<U;V>; <NULL<U> ¬ 0;
17        "T" ¬ ADD<MUL<CAR<U>; CAR<V>>; DOT<CDR<U>; CDR<V>>>>>;
18
19    'NULL' &<<L>; EQ<L; "NIL">>>>
20
21    *SAMPLE INPUTS
22    <((1 1 1 1) (1 1 1 1) (1 1 1 1) (1 1 1 1));
23     ((1 1 1 1) (1 1 1 1) (1 1 1 1) (1 1 1 1))>
24    #
```

```
****** EVALUATION BEGINS ******

RESULT IS:
(((4 4 4 4) (4 4 4 4) (4 4 4 4) (4 4 4 4)) 1 2 2 4 4 10 8 20 14 32 20 46

 26 53 29 53 29 45 25 31 19 19 13 10 8 5 5 3 3 2 2 1 1 1 1 1 1)


PROCESSORS REQUIRED FOR OPTIMUM PARALLELING.            53
EXECUTION STEPS (PARALLEL).................             37
EXECUTION STEPS (SEQUENTIAL)...............            549
SPEED-UP RATIO (SEQUENTIAL/PARALLEL).......      14.83784
FREE STORAGE REMAINING:         11273
019.56 SECONDS IN EXECUTION
```

Program 3.

```
$$
$A          GRAPHING ARITHMETIC OPERATIONS ONLY
    1       *       THIS FUNCTION PERFORMS MATRIX MULTIPLICATION.
    2       *       DOT PRODUCTS ARE COMPUTED BY PERFORMING SEQUENTIAL
    3       *       ADDITIONS OF INTEGER PRODUCTS.
    4
$L          SUPPRESS FUNCTION LISTING
$L           TURN ON LISTING FOR ARGUMENTS
   21       *SAMPLE INPUTS
   22       <((1 1 1 1) (1 1 1 1) (1 1 1 1) (1 1 1 1));
   23        ((1 1 1 1) (1 1 1 1) (1 1 1 1) (1 1 1 1))>
   24       #

****** EVALUATION BEGINS ******

RESULT IS:
(((4 4 4 4) (4 4 4 4) (4 4 4 4) (4 4 4 4)) 64 16 16 16 16)


PROCESSORS REQUIRED FOR OPTIMUM PARALLELING.            64
EXECUTION STEPS (PARALLEL).....                          5
EXECUTION STEPS (SEQUENTIAL)...                        128
SPEED-UP RATIO (SEQUENTIAL/PARALLEL).....          25.59999
FREE STORAGE REMAINING:         11369
016.35 SECONDS IN EXECUTION
```

Program 4.

```
  1     *       THIS FUNCTION PERFORMS MATRIX MULTIPLICATION.
  2     *       DOT PRODUCTS ARE COMPUTED BY PERFORMING PARALLEL
  3     *       ADDITIONS OF PAIRS OF INTEGER PRODUCTS.
  4
  5     &<<A;B>;
  6         &<<MATMUL; ROW; DOT; SUM; REDUCE; VMUL; CADR; CDDR; NULL>;
  7
  8             MATMUL<A;B>>
  9
 10     <'MATMUL' &<<X;Y>; <NULL<X> ¬ "NIL";
 11         "T" ¬ CONS<ROW<CAR<X>; Y>; MATMUL<CDR<X>; Y>>>>>;
 12
 13     'ROW' &<<R;C>; <NULL<C> ¬ "NIL";
 14         "T" ¬ CONS<DOT<R; CAR<C>>; ROW<R; CDR<C>>>>>;
 15
 16     'DOT' &<<U;V>; SUM<VMUL<U;V>>>;
 17
 18     'SUM' &<<A>; <NULL<CDDR<A>> ¬ ADD<CAR<A>; CADR<A>>;
 19         "T" ¬ SUM<REDUCE<A>>>>;
 20
 21     'REDUCE' &<<A>; <NULL<A> ¬ "NIL"; NULL<CDR<A>> ¬ A;
 22         "T" ¬ CONS<ADD<CAR<A>; CADR<A>>; REDUCE<CDDR<A>>>>>;
 23
 24     'VMUL' &<<U;V>; <NULL<V> ¬ "NIL";
 25         "T" ¬ CONS<MUL<CAR<U>; CAR<V>>; VMUL<CDR<U>; CDR<V>>>>>;
 26
 27     'CADR' &<<L>; CAR<CDR<L>>>;
 28
 29     'CDDR' &<<L>; CDR<CDR<L>>>;
 30
 31     'NULL' &<<L>; EQ<L; "NIL">>>>
 32
 33     *SAMPLE INPUTS
 34     <<(1 1 1 1) (1 1 1 1) (1 1 1 1) (1 1 1 1));
 35      ((1 1 1 1) (1 1 1 1) (1 1 1 1) (1 1 1 1))>
 36     #

****** EVALUATION BEGINS ******

RESULT IS:
(((4 4 4 4) (4 4 4 4) (4 4 4 4) (4 4 4 4)) 1 2 2 4 4 10 8 20 14 32 20 4

 26 53 29 53 29 46 26 34 22 25 19 21 18 21 18 24 21 28 25 28 27 26 29 2

 28 20 25 17 22 14 18 10 13 7 9 5 6 3 3 2 2 1 1 1 1 1 1)


PROCESSORS REQUIRED FOR OPTIMUM PARALLELING.            53
EXECUTION STEPS (PARALLEL).....                         59
EXECUTION STEPS (SEQUENTIAL).....                     1045
SPEED-UP RATIO (SEQUENTIAL/PARALLEL).....         17.71185
FREE STORAGE REMAINING:          6954
038.90 SECONDS IN EXECUTION
```

Program 5.

```
$$
$A          GRAPHING ARITHMETIC OPERATIONS ONLY
    1
    2       *       THIS FUNCTION PERFORMS MATRIX MULTIPLICATION.
    3       *       DOT PRODUCTS ARE COMPUTED BY PERFORMING PARALLEL
    4       *       ADDITIONS OF PAIRS OF INTEGER PRODUCTS.
    5
$L          SUPPRESS FUNCTION LISTING
$L          LIST ARGUMENTS
   34       *SAMPLE INPUTS
   35       <((1 1 1 1) (1 1 1 1) (1 1 1 1) (1 1 1 1));
   36        ((1 1 1 1) (1 1 1 1) (1 1 1 1) (1 1 1 1))>
   37       #                                          .

****** EVALUATION BEGINS ******

RESULT IS:
(((4 4 4 4) (4 4 4 4) (4 4 4 4) (4 4 4 4)) 64 32 16)


PROCESSORS REQUIRED FOR OPTIMUM PARALLELING.            64
EXECUTION STEPS (PARALLEL).................              3
EXECUTION STEPS (SEQUENTIAL)...............            112
SPEED-UP RATIO (SEQUENTIAL/PARALLEL).......       37.33333
FREE STORAGE REMAINING:          7122
031.96 SECONDS IN EXECUTION
```

Program 6.

```
1     *      DIFFERENTIATE A POLYNOMIAL WITH RESPECT TO
2     *      ONE OF ITS VARIABLES.  SIMPLIFY THE RESULT.
3
4     &<<P;X>;  'SIMPLIFY DERIVATIVE OF "P" W.R.T. "X"'
5
6        &<<CADR; CADDR; CONS3>;  'DEFINE PRIMITIVES ON A-LIST'
7
8           @<SIMP; &<<P>;  'SIMPLIFY FUNCTION'
9
10             <ATOM<P> ¬ P;
11             "T" ¬
12                &<<P1; OP; P2>;
13
14                   <EQ<OP; "+"> ¬
15                      <EQ<P1; "0"> ¬ P2;
16                      EQ<P2; "0"> ¬ P1;
17                      "T" ¬ CONS3<P1; "+"; P2>
18                      >;
19                   "T" ¬
20                      <EQ<P1; "0"> ¬ "0";
21                      EQ<P2; "0"> ¬ "0";
22                      EQ<P1; "1"> ¬ P2;
23                      EQ<P2; "1"> ¬ P1;
24                      "T" ¬ CONS3<P1; "*"; P2>
25                      >
26                   >
27                >
28                <SIMP<CAR<P>>; CADR<P>; SIMP<CADDR<P>>>
29             >
30          >>
31
32          <  'ARGUMENT FOR SIMP'
33          @<DIFF; &<<P; X>;  'DERIVATIVE FUNCTION'
34
35             <ATOM<P> ¬ <EQ<P; X> ¬ "1"; "T" ¬ "0">;
36             "T" ¬ &<<P1; OP; P2>; 'MORE THAN ONE TERM'
37
38                <EQ<OP; "+"> ¬
39                   CONS3<DIFF<P1; X>; "+"; DIFF<P2; X>>;
40                "T" ¬ CONS3<CONS3<DIFF<P1; X>; "*"; P2>; "+";
41                   CONS3<DIFF<P2; X>; "*"; P1>>>>
42
43                <CAR<P>; CADR<P>; CADDR<P>>>>>
44
45          <P; X>  'ARGUMENTS FOR DIFF'
46          >  'END OF SIMP ARGUMENT'
47       >
48
49       <  'PRIMITIVE DEFINITIONS'
50       &<<X>; CAR<CDR<X>>>;
51       &<<X>; CADR<CDR<X>>>;
52       &<<X; Y; Z>; CONS<X; CONS<Y; CONS<Z; "NIL">>>>
53       >  'END PRIMITIVE DEFINITIONS'
54    >  'END.'
55
56    *      SAMPLE ARGUMENTS WITH SYMMETRICALLY ORGANIZED
57    *      FOURTH-DEGREE BINOMIAL.
58
59    <(((X + Y) * (X + Y)) * ((X + Y) * (X + Y))); X>
60    #
```

```
****** EVALUATION BEGINS ******

RESULT IS:
( (((((X + Y) + (X + Y)) * ((X + Y) * (X + Y))) + (((X + Y) + (X + Y)) * (

(X + Y) * (X + Y)))) 1 3 2 1 1 2 6 4 2 2 4 12 8 4 4 8 8 4 4 4 4 4 4 2 2

2 2 2 2 1 1 1 1 3 3 4 4 7 7 11 11 18 18 23 23 20 20 14 13 11 9 8 7 6 6 6

 6 5 5 4 4 3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1)


PROCESSORS REQUIRED FOR OPTIMUM PARALLELING.            23
EXECUTION STEPS (PARALLEL).....                         81
EXECUTION STEPS (SEQUENTIAL)...............            422
SPEED-UP RATIO (SEQUENTIAL/PARALLEL).....          5.209876
FREE STORAGE REMAINING:          11739
016.96 SECONDS IN EXECUTION
```

Program 7.

```
$$
   1      *       DIFFERENTIATE A POLYNOMIAL WITH RESPECT TO
   2      *       ONE OF ITS VARIABLES.  SIMPLIFY THE RESULT.
   3
$L       SUPRESS FUNCTION LISTING
$L       TURN ON LISTING FOR ARGUMENTS
  56      *       SAMPLE ARGUMENTS WITH ASYMMETRICALLY ORGANIZED
  57      *       FOURTH-DEGREE BINOMIAL.
  58
  59      <((X + Y) * ((X + Y) * ((X + Y) * (X + Y)))); X>
  60      #


****** EVALUATION BEGINS ******

RESULT IS:
((((X + Y) * ((X + Y) * (X + Y))) + ((((X + Y) * (X + Y)) + (((X + Y) +

(X + Y)) * (X + Y))) * (X + Y))) 1 3 2 1 1 2 6 4 2 2 4 8 5 3 3 5 9 6 3 3

 5 5 3 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 3 3 4 4 7 7 8 8 12 12

13 12 14 14 18 18 17 16 13 13 11 11 8 7 5 4 4 4 4 4 4 4 3 3 3 3 3 3 2 2

2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1)


PROCESSORS REQUIRED FOR OPTIMUM PARALLELING.          18
EXECUTION STEPS (PARALLEL).....                      112
EXECUTION STEPS (SEQUENTIAL)...                      452
SPEED-UP RATIO (SEQUENTIAL/PARALLEL).....       4.035714
FREE STORAGE REMAINING:          11382
017.76 SECONDS IN EXECUTION
```

Program 8.

```
1     * EVALQUOTE
2
3     &<<FN; ARGS>;
4
5         'SUB-FUNCTION NAMES'
6         &<<APPLY; EVAL; EVCON; EVLIS; PAIRLIS; ASSOC;
7            NULL; CAAR; CADR; CDAR; CADDR; CADAR>;
8
9             'DEFINITION OF EVALQUOTE'
10            APPLY<FN; ARGS; "NIL">>
11
12        'SUB-FUNCTION DEFINITIONS'
13
14        <'APPLY' &<<FN; X; A>;
15            <ATOM<FN> ¬
16               <EQ<FN; "CAR"> ¬ CAAR<X>;
17                EQ<FN; "CDR"> ¬ CDAR<X>;
18                EQ<FN; "CONS"> ¬ CONS<CAR<X>; CADR<X>>;
19                EQ<FN; "ATOM"> ¬ ATOM<CAR<X>>;
20                EQ<FN; "EQ"> ¬ EQ<CAR<X>; CADR<X>>;
21                "T" ¬ APPLY<EVAL<FN;A>; X; A>>;
22            EQ<CAR<FN>; "LAMBDA"> ¬
23                EVAL<CADDR<FN>; PAIRLIS<CADR<FN>; X; A>>;
24            EQ<CAR<FN>; "LABEL"> ¬
25                APPLY<CADDR<FN>; X;
26                    CONS<CONS<CADR<FN>; CADDR<FN>>; A>>>>;
27
28        'EVAL' &<<E; A>;
29            <ATOM<E> ¬ CDR<ASSOC<E; A>>;
30             ATOM<CAR<E>> ¬
31                <EQ<CAR<E>; "QUOTE"> ¬ CADR<E>;
32                 EQ<CAR<E>; "COND"> ¬ EVCON<CDR<E>; A>;
33                 "T"¬ APPLY<CAR<E>; EVLIS<CDR<E>; A>; A>>;
34            "T" ¬ APPLY<CAR<E>; EVLIS<CDR<E>; A>; A>>>;
35
36        'EVCON' &<<C; A>; <NULL<C> ¬ "UNDEFINED";
37                          EVAL<CAAR<C>; A> ¬ EVAL<CADAR<C>; A>;
38            "T" ¬ EVCON<CDR<C>; A>>>;
39
40        'EVLIS' &<<L; A>;
41            <NULL<L> ¬ "NIL";
42             "T" ¬ CONS<EVAL<CAR<L>; A>; EVLIS<CDR<L>; A>>>>;
43
44        'PAIRLIS' &<<X; Y; A>;
45            <NULL<X> ¬ A; "T" ¬ CONS<CONS<CAR<X>; CAR<Y>>;
46                PAIRLIS<CDR<X>; CDR<Y>; A>>>>;
47
48        'ASSOC' &<<X; A>;
49            <EQ<CAAR<A>; X> ¬ CAR<A>; "T" ¬ ASSOC<X; CDR<A>>>>;
50
51        'NULL' &<<L>; EQ<L; "NIL">>;
52
53        'CAAR' &<<L>; CAR<CAR<L>>>;
54
55        'CADR' &<<L>; CAR<CDR<L>>>;
56
57        'CDAR' &<<L>; CDR<CAR<L>>>;
58
59        'CADDR' &<<L>; CADR<CDR<L>>>;
60
61        'CADAR' &<<L>; CAR<CDAR<L>>>>>
```

```
63
64      * SAMPLE ARGUMENTS FOR EVALQUOTE
65
66      <(LABEL PAIRLIS (LAMBDA (X Y A)
67         (COND ((EQ X (QUOTE NIL)) A)
68          ((QUOTE T) (CONS (CONS (CAR X) (CAR Y))
69                           (PAIRLIS (CDR X) (CDR Y) A))))));
70       ((A B) (1 2) ((C.3)))>
71      #

****** EVALUATION BEGINS ******

RESULT IS:
(((A.1) (B.2) (C.3)) 1 1 1 1 1 3 3 2 1 1 1 1 1 2 2 2 4 2 4 2 4 2 1 1 1

 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 2 2 3 3 2 2 2 1 1 1 1 1 1 1 1 1

 1 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 2 2 3 3 2 2 2 2

 2 4 3 5 5 5 5 6 6 6 5 8 6 8 7 6 6 5 5 5 5 5 5 3 3 3 2 2 2 2 2 2 2 2 2

 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 4 2 4 2 4 2 1

 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 2 2 3 3 2 2 2 1 1 1 1 1 1

 1 1 1 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 2 2 3 3 2 2

 2 3 2 4 3 5 5 5 5 6 6 6 5 8 6 8 7 6 6 5 5 5 5 5 5 3 3 3 2 2 2 2 2 2 2

 2 3 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

 1 1 1 2 2 2 4 2 4 2 4 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1

 2 3 3 2 2 2 1 1 1 1 1 1 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

 1 1 1 1 1 1 1 2 1 1 1 1 1 1 1 1 2 1 1)


PROCESSORS REQUIRED FOR OPTIMUM PARALLELING.            8
EXECUTION STEPS (PARALLEL).....                       405
EXECUTION STEPS (SEQUENTIAL)......                    754
SPEED-UP RATIO (SEQUENTIAL/PARALLEL).....        1.861728
FREE STORAGE REMAINING:          8349
032.12 SECONDS IN EXECUTION
```

APPENDIX E

ALGOL-W INTERPRETER

```
0000 1-     BEGIN COMMENT**************************************************
0001 --     *                                                             *
0001 --     *                                                             *
0001 --     *                                                             *
0001 --     *                          EVALQUOTE2                         *
0001 --     *                                                             *
0001 --     *                                                             *
0001 --     *                                                             *
0001 --     *                                                             *
0001 --     *     EVALQUOTE2 IS AN ALGOL-W PROGRAM DESIGNED TO            *
0001 --     *     TRANSLATE AND INTERPRET INPUT PROGRAMS WRITTEN          *
0001 --     *     IN THE META-LANGUAGE OF PURE LISP.  DURING              *
0001 --     *     INTERPRETATION, EVALQUOTE2 ANALYSES THE DATA FLOW       *
0001 --     *     THROUGH THE LISP PROGRAM AND GENERATES A VECTOR         *
0001 --     *     DESCRIBING THE DATA FLOW GRAPH.                         *
0001 --     *                                                             *
0001 --     *                                                             *
0001 --     *                        ORGANIZATION                         *
0001 --     *                                                             *
0001 --     *     THE PROGRAM HAS BEEN LOGICALLY DIVIDED INTO ELEVEN      *
0001 --     *     SECTIONS.                                               *
0001 --     *                                                             *
0001 --     *                                                             *
0001 --     *     I :    GLOBAL DECLARATIONS                              *
0001 --     *                                                             *
0001 --     *     II :   PRIMITIVES                                       *
0001 --     *                                                             *
0001 --     *     III :  STORAGE MANAGEMENT                               *
0001 --     *                                                             *
0001 --     *     IV :   INPUT SUPPORT                                    *
0001 --     *                                                             *
0001 --     *     V :    PROPERTY LIST ACCESS                             *
0001 --     *                                                             *
0001 --     *     VI :   INITIALIZATION                                   *
0001 --     *                                                             *
0001 --     *     VII :  TRANSLATION                                      *
0001 --     *                                                             *
0001 --     *     VIII:  INTERPRETATION                                   *
0001 --     *                                                             *
0001 --     *     IX :   ERROR HANDLING                                   *
0001 --     *                                                             *
0001 --     *     X :    OUTPUT                                           *
0001 --     *                                                             *
0001 --     *     XI :   MONITOR                                          *
0001 --     *                                                             *
0001 --     *************************************************************;
0001 --
0001 --
0001 --     COMMENT*******************************************************
0001 --     *                                                             *
0001 --     *     SECTION I:  GLOBAL DECLARATIONS                         *
0001 --     *                                                             *
0001 --     *************************************************************;
0001 --
0001 --     INTEGER FLH;        COMMENT FREE LIST HEADER;
0002 --     INTEGER LENGTH;     COMMENT THE LENGTH OF THE CURRENT TOKEN;
0003 --     INTEGER IBP;        COMMENT THE INPUT BUFFER POINTER;
0004 --     INTEGER LINE_NO;    COMMENT THE INPUT LINE NUMBER;
0005 --
0005 --     COMMENT  THE FOLLOWING ARE THE LOCATIONS OF THE ATOM HEADER
0005 --              CELLS OF THE RESERVED WORDS OF THE LISP LANGUAGE;
0005 --
0005 --     INTEGER LAMBDA, LABEL, COND, QUOTE;
0006 --     INTEGER ICAR, ICDR, ICONS, IEQ, IATOM, IADD, IMUL;
0007 --     INTEGER PNAME, T, F, NIL;
0008 --
0008 --     INTEGER CONSTANT, IDENTIFIER, SPECIAL, NUMS;
0009 --         COMMENT TOKEN CATEGORIES;
0009 --     INTEGER TYPE;   COMMENT TOKEN CATEGORY OF CURRENT TOKEN;
0010 --     INTEGER HEADER;  COMMENT PTR TO ATOM HEADER CELL OF TOK;
0011 --
0011 --     STRING(1) C, NC;
0012 --         COMMENT CURRENT AND NEXT CHARACTER IN INPUT STREAM;
0012 --     STRING(80) BUF;     COMMENT THE INPUT BUFFER;
0013 --     STRING(72) TOK;    COMMENT THE CURRENT TOKEN;
0014 --
0014 --     LOGICAL FLAGS, LIST, TRANS, ARITH;
0015 --         COMMENT COMPILER TOGGLES;
0015 --     LOGICAL INTRANS;
0016 --         COMMENT INDICATES IF IN TRANSLATION OR INTERPRETATION;
0016 --     LOGICAL COLLECTED;
0017 --         COMMENT INDICATES IF HASH TABLE HAS BEEN FREED;
0017 --
0017 --     BITS ARRAY M(0::16383);  COMMENT WORKSPACE;
0018 --
```

```
0018 --       COMMENT**********************************************************
0018 --       *                                                                *
0019 --       *      SECTION II:  PRIMITIVES                                   *
0018 --       *                                                                *
0018 --       ******************************************************************;
0018 --
0018 --       PROCEDURE SETCDR(INTEGER VALUE X, Y);
0019 --           COMMENT SET THE D REGISTER OF CELL AT X TO VALUE Y;
0019 --           M(X) := (M(X) AND #FFFF0000) OR BITSTRING(Y);
0020 --
0020 --       PROCEDURE SETCAR(INTEGER VALUE X, Y);
0021 --           COMMENT SET THE A REGISTER OF CELL AT X TO VALUE Y;
0021 --           M(X) := (M(X) AND #FFFF) OR (BITSTRING(Y) SHL 16);
0022 --
0022 --       INTEGER PROCEDURE CDR(INTEGER VALUE N);
0023 --           COMMENT EXTRACT THE D REGISTER CONTENTS;
0023 --           NUMBER(M(N) AND #FFFF);
0024 --
0024 --       INTEGER PROCEDURE CAR(INTEGER VALUE N);
0025 --           COMMENT EXTRACT THE A REGISTER CONTENTS;
0025 --           NUMBER(M(N) SHR 16);
0026 --
0026 --       INTEGER PROCEDURE CONS(INTEGER VALUE X, Y);
0027 --           COMMENT ADD X TO FRONT OF LIST AT Y;
0027 2-           BEGIN INTEGER A;
0029 --           A := ALLOCATE;
0030 --           SETCAR(A,X);   SETCDR(A,Y);
0032 --           A
0032 -2           END CONS;
0033 --
0033 --       LOGICAL PROCEDURE EQ(INTEGER VALUE X, Y);
0034 --           X = Y;
0035 --
0035 --       LOGICAL PROCEDURE ATOM(INTEGER VALUE X);
0036 --           COMMENT RETURN TRUE IF CAR(X) IS CHARACTER ATOM CELL;
0036 --           M(X) SHR 17 = #00007FFF;
0037 --
0037 --       INTEGER PROCEDURE ADD (INTEGER VALUE X, Y);
0038 --           COMMENT RETURN POINTER TO NEW CONSTANT ATOM CONTAINING
0038 --                   SUM OF CONSTANT ATOMS X AND Y;
0038 2-           BEGIN
0039 --           X := NUMBER(M(CDR(X)));
0040 --           Y := NUMBER(M(CDR(Y)));
0041 --           BUILD_C_ATOM(X + Y)
0041 -2           END ADD;
0042 --
0042 --       INTEGER PROCEDURE MUL (INTEGER VALUE X, Y);
0043 --           COMMENT RETURN POINTER TO NEW CONSTANT ATOM CONTAINING
0043 --                   PRODUCT OF CONSTANT ATOMS X AND Y;
0043 2-           BEGIN
0044 --           X := NUMBER(M(CDR(X)));
0045 --           Y := NUMBER(M(CDR(Y)));
0046 --           BUILD_C_ATOM(X * Y)
0046 -2           END MUL;
0047 --
0047 --       INTEGER PROCEDURE CAAR (INTEGER VALUE X);
0048 --           CAR(CAR(X));
0049 --
0049 --       INTEGER PROCEDURE CDAR (INTEGER VALUE X);
0050 --           CDR(CAR(X));
0051 --
0051 --       INTEGER PROCEDURE CADR (INTEGER VALUE X);
0052 --           CAR(CDR(X));
0053 --
0053 --       INTEGER PROCEDURE CADDR (INTEGER VALUE X);
0054 --           CADR(CDR(X));
0055 --
0055 --       INTEGER PROCEDURE CADAR (INTEGER VALUE X );
0056 --           CAR(CDAR(X));
0057 --
0057 --       INTEGER PROCEDURE CONS3(INTEGER VALUE X,Y,Z);
0058 --           COMMENT PLACE X AND Y ON LIST AT Z;
0058 --           CONS(X,CONS(Y,CONS(Z,NIL)));
0059 --
0059 --       LOGICAL PROCEDURE NUL (INTEGER VALUE X);
0060 --           COMMENT IS LIST EXHAUSTED;
0060 --           X = NIL;
0061 --
```

```
0061 --       COMMENT************************************************************
0061 --       *                                                                   *
0061 --       *      SECTION III:  STORAGE MANAGEMENT                             *
0061 --       *                                                                   *
0061 --       *********************************************************************;
0061 --
0061 --       INTEGER PROCEDURE ALLOCATE;
0062 --           COMMENT ALLOCATE ONE CELL;
0062 2-           BEGIN
0063 --           INTEGER X;
0064 --           X := CDR(FLH);
0065 --           IF X = NIL THEN
0065 3-               BEGIN
0066 --               IF INTRANS OR COLLECTED THEN
0066 --                   ERROR(2)
0066 --               ELSE
0067 4-                   BEGIN
0068 --                   FREE_HASH_TABLE;
0069 --                   X := CDR(FLH);
0070 -4                   END;
0071 -3               END;
0072 --           SETCDR(FLH, CDR(X));
0073 --           X
0073 -2           END ALLOCATE;
0074 --
0074 --       PROCEDURE FREE(INTEGER VALUE X);
0075 --           COMMENT RELEASE CELL AT X TO FREE LIST;
0075 2-           BEGIN
0076 --           M(X):=M(FLH);
0077 --           M(FLH):=BITSTRING(X);
0078 -2           END FREE;
0079 --
0079 --       PROCEDURE FREE_A_LIST(INTEGER VALUE P,A);
0080 --           COMMENT FREE OUTDATED PORTION OF A_LIST;
0080 --           IF P ¬= A THEN
0080 2-               BEGIN
0081 --               FREE_A_LIST(CDR(P),A);
0082 --               FREE(CAR(P));
0083 --               FREE(P);
0084 -2               END;
0085 --
0085 --       PROCEDURE FREE_HASH_TABLE;
0086 2-           BEGIN
0087 --
0087 --           PROCEDURE FREE_STACK(INTEGER VALUE I);
0088 --               IF I ¬= NIL THEN
0088 3-                   BEGIN
0089 --                   FREE_STACK(CDR(I));
0090 --                   FREE(I);
0091 -3                   END;
0092 --
0092 --           FOR I := 26 UNTIL 153 DO
0092 3-               BEGIN
0093 --               FREE_STACK(CDR(I));
0094 --               FREE_STACK(CAR(I));
0095 --               FREE(I);
0096 -3               END;
0097 --           COLLECTED := TRUE;
0098 -2           END FREE_HASH_TABLE;
0099 --
0099 --       INTEGER PROCEDURE FREE_CELLS;
0100 --           COMMENT RETURN SIZE OF FREE STORAGE;
0100 2-           BEGIN INTEGER PTR, COUNT;
0102 --           PTR := CDR(FLH); COUNT := 0;
0104 --           WHILE PTR ¬= NIL DO
0104 3-               BEGIN
0105 --               COUNT := COUNT + 1;
0106 --               PTR := CDR(PTR);
0107 -3               END;
0108 --           COUNT
0108 -2           END FREE_CELLS;
0109 --
0109 --       COMMENT************************************************************
0109 --       *                                                                   *
0109 --       *      SECTION IV:  INPUT SUPPORT                                   *
0109 --       *                                                                   *
0109 --       *********************************************************************;
0109 --
0109 --       INTEGER PROCEDURE BUILD_C_ATOM(INTEGER VALUE A);
0110 --           COMMENT CREATE ATOM-HEADER STRUCTURE FOR CONSTANT A;
0110 2-           BEGIN
0111 --           INTEGER C;
0112 --           C := ALLOCATE;
0113 --           M(C) := BITSTRING(A);
0114 --           CONS(NUMBER(#FFFE),C)
0114 -2           END BUILD_C_ATOM;
0115 --
0115 --
```

```
0115 --       PROCEDURE BUILD_ATOM (STRING(72) VALUE WORD;
0116 --             INTEGER VALUE A,B);
0117 --           COMMENT CREATE ATOM HEADER STRUCTURE FOR WORD OF
0117 --             LENGTH A AT LOCATION B;
0117 2-           BEGIN
0118 --
0118 --           INTEGER PROCEDURE BUILD_PNAME (STRING(72) VALUE WORD;
0119 --                 INTEGER VALUE A,P);
0120 --               COMMENT CONSTRUCT CELLS TO HOLD CHARACTERS OF WORD
0120 --                 STARTING AT LOCATION P FOR 4  LETTERS OR UNTIL A;
0120 3-               BEGIN
0121 --               INTEGER B,C;
0122 --
0122 --               PROCEDURE INSERT_CHAR(STRING(1) VALUE LTR;
0123 --                                     INTEGER VALUE A);
0124 --                   COMMENT INSERT LTR IN CELL A;
0124 --                   M(A):=(M(A) SHL 8) OR BITSTRING (DECODE(LTR));
0125 --
0125 --               B:=ALLOCATE;
0126 --               C:=ALLOCATE;
0127 --               SETCAR(B,C);
0128 --               M(C):=#0;
0129 --               IF (A-P) < 5 THEN
0129 4-                 BEGIN
0130 --                 SETCDR(B,NIL);
0131 --                 FOR I:=P UNTIL (A-1) DO  INSERT_CHAR(WORD(I|1),C);
0132 --                 FOR I:=1 UNTIL (4+P-A) DO  M(C):=M(C)SHL 8;
0133 -4                END
0133 --               ELSE
0134 4-                 BEGIN
0135 --                 FOR I:=P UNTIL (P+3) DO  INSERT_CHAR(WORD(I|1),C);
0136 --                 P:=P+4;
0137 --                 SETCDR(B,BUILD_PNAME(WORD,A,P));
0138 -4                 END;
0139 --               B
0139 -3               END BUILD_PNAME;
0140 --
0140 --           IF A <= 0 THEN  ERROR(3);
0141 --           M(B) := M(B) OR #FFFF0000;
0142 --           SETCDR(B,CONS(PNAME,CONS(BUILD_PNAME(WORD,A,0),NIL)))
0142 -2           END BUILD_ATOM;
0143 --
0143 --       INTEGER PROCEDURE HASH (INTEGER VALUE ACCL;
0144 --             STRING(72)VALUE ACCUM);
0145 --           COMMENT COMPUTE AND RETURN HASHED VALUE OF TOK IN
0145 --             ACCUMULATOR GIVEN TOK LENGTH;
0145 2-           BEGIN
0146 --           INTEGER SUM,H;
0147 --           SUM:=0;
0148 --           FOR I:=0 UNTIL ACCL-1 DO  SUM:=SUM+DECODE(ACCUM(I|1));
0149 --           H:= 26+SUM REM 128;
0150 --           H
0150 -2           END HASH;
0151 --
0151 --       PROCEDURE PUSH(INTEGER VALUE X, I);
0152 --           COMMENT PUSH ATOM ONTO HASH TABLE;
0152 --           SETCDR(I,CONS(X,CDR(I)));
0153 --
0153 --       COMMENT************************************************
0153 --       *                                                      *
0153 --       *      SECTION 7:  PROPERTY LIST ACCESS                *
0153 --       *                                                      *
0153 --       ************************************************************;
0153 --
0153 --       INTEGER PROCEDURE FIND_VALUE(INTEGER VALUE HEADER,ATTRIB);
0154 --           COMMENT SCAN PROPERTY LIST OF ATOM AT HEADER FOR
0154 --             VALUE OF ATTRIB;
0154 2-           BEGIN
0155 --           INTEGER P;
0156 --           P := CDR(HEADER);
0157 --           WHILE CAR(P) ¬= ATTRIB DO
0157 3-               BEGIN
0158 --               IF CDR(P) = NIL THEN  ERROR(1);
0159 --               P := CDR(CDR(P));
0160 --               IF P = NIL THEN  ERROR(1);
0161 -3               END;
0162 --           CADR(P)
0162 -2           END FIND_VALUE;
0163 --
0163 --
0163 --       PROCEDURE GET_PNAME(INTEGER VALUE X; STRING(72) RESULT BUFF;
0165 --             INTEGER RESULT LENGTH);
0166 --           COMMENT GET PRINTNAME AND LENGTH OF ATOM AT X;
0166 2-           BEGIN
0167 --
0167 --           PROCEDURE GET_CHARS(INTEGER VALUE X);
0168 --               COMMENT EXTRACT CHARACTERS FROM CELL AT X;
0168 --               FOR J:=0 STEP 8 UNTIL 24 DO
0168 3-                   BEGIN INTEGER N;
0170 --                   N := NUMBER((M(X) SHL J) SHR 24);
0171 --                   IF N > 0 THEN
0171 4-                       BEGIN
0172 --                       BUFF(LENGTH|1) := CODE(N);
0173 --                       LENGTH := LENGTH+1;
0174 -4                       END;
0175 -3                   END GET_CHARS;
```

```
0176 --          LENGTH := 0;
0176 --          BUFF := " ";
0177 --          X := FIND_VALUE(X,PNAME);
0178 --          WHILE X ¬= NIL DO
0179 --              BEGIN
0179 3-              GET_CHARS(CAR(X));
0180 --              X := CDR(X);
0181 --              END;
0182 -3          END GET_PNAME;
0183 -2
0184 --      COMMENT************************************************
0184 --      *                                                     *
0184 --      *      SECTION VI:  INITIALIZATION                    *
0184 --      *                                                     *
0184 --      ******************************************************;
0184 --
0184 --      PROCEDURE INITIALIZE;
0185 --          COMMENT INITIALIZE MEMORY SPACE;
0185 2-          BEGIN
0186 --
0186 --          PROCEDURE SETRESWDS;
0187 --              COMMENT INITIALIZE RESERVED WORDS AND
0187 --                  PREDEFINED FUNCTIONS;
0187 3-              BEGIN
0188 --
0188 --              PROCEDURE SETWORD(STRING(72) VALUE WORD;
0189 --                      INTEGER VALUE LENGTH, ADDR);
0190 --                  COMMENT PLACE RESERVED WORDS IN HASH TABLE AND
0190 --                      BUILD ATOM HEADER STRUCTURE AT ADDR;
0190 4-                  BEGIN
0191 --                  INTEGER VAL;
0192 --                  VAL := HASH(LENGTH, WORD);
0193 --                  COMMENT PLACE RESWORDS ON CAR SIDE OF HASH TABLE;
0193 --                  SETCAR(VAL,CONS(ADDR,CAR(VAL)));
0194 --                  BUILD_ATOM(WORD, LENGTH, ADDR);
0195 -4                  END SETWORD;
0196 --
0196 --              SETWORD("PNAME" ,5,1);
0197 --              SETWORD("T"     ,1,5);
0198 --              SETWORD("F"     ,1,6);
0199 --              SETWORD("NIL"   ,3,7);
0200 --              SETWORD("LAMBDA",6,8);
0201 --              SETWORD("LABEL" ,5,9);
0202 --              SETWORD("COND"  ,4,10);
0203 --              SETWORD("QUOTE" ,5,11);
0204 --              SETWORD("CAR"   ,3,12);
0205 --              SETWORD("CDR"   ,3,13);
0206 --              SETWORD("CONS"  ,4,14);
0207 --              SETWORD("EQ"    ,2,15);
0208 --              SETWORD("ATOM"  ,4,16);
0209 --              SETWORD("MUL"   ,3,17);
0210 --              SETWORD("ADD"   ,3,18);
0211 -3              END SETRESWDS;
0212 --
0212 --          C := NC := " ";
0213 --          PLH := 0;  LINE_NO := 0;  IBP := 80;
0216 --          LIST := TRUE;
0217 --          COLLECTED := FLAGS := TRANS := ARITH := FALSE;
0218 --
0218 --          COMMENT INITIALIZE SYMBOLS FOR TOKEN CATEGORIES;
0218 --          NUMS       := 0;
0219 --          IDENTIFIER := 1;
0220 --          SPECIAL    := 2;
0221 --          CONSTANT   := 3;
0222 --
0222 --          COMMENT INITIALIZE RESERVED WORD LOCATIONS;
0222 --          PNAME  := 1;
0223 --          T      := 5;
0224 --          F      := 6;
0225 --          NIL    := 7;
0226 --          LAMBDA := 8;
0227 --          LABEL  := 9;
0228 --          COND   := 10;
0229 --          QUOTE  := 11;
0230 --          ICAR   := 12;
0231 --          ICDR   := 13;
0232 --          ICONS  := 14;
0233 --          IEQ    := 15;
0234 --          IATOM  := 16;
0235 --          IMUL   := 17;
0236 --          IADD   := 18;
0237 --
0237 --          COMMENT PLACE NILS ON BOTH SIDES OF HASH TABLE;
0237 --          FOR I:=26 UNTIL 153 DO  M(I) := #00070007;
0238 --
0238 --          COMMENT INITIALIZE THE FREE LIST;
0238 --          FOR I:=154 UNTIL 16382 DO  M(I) := BITSTRING(I+1);
0239 --          M(16383) :=#7;
0240 --          M(0) := BITSTRING(154);
0241 --          SCANNER;
0242 --          SETRESWDS;
0243 -2          END INITIALIZE;
0244 --
```

```
0244 --       COMMENT**********************************************************
0244 --       *                                                                *
0244 --       *      SECTION VII:  SCANNER                                     *
0244 --       *                                                                *
0244 --       ******************************************************************;
0244 --
0244 --       PROCEDURE SCANNER;
0245 --           COMMENT SCAN INPUT STREAM FOR NEXT TOK, ASSIGN
0245 --               TYPE, AND BUILD ATOM HEADER CELL IF REQUIRED;
0245 2-           BEGIN
0246 --
0246 --           PROCEDURE GNC;
0247 --               COMMENT GET NEXT CHARACTER FROM INPUT;
0247 3-               BEGIN
0248 --
0248 --               PROCEDURE BUMP_IBP;
0249 --                   COMMENT INPUT BUFFER POINTER;
0249 4-                   BEGIN
0250 --
0250 --                   PROCEDURE READBUF;
0251 --                       COMMENT INPUTS NEXT RECORD, OUTPUTS
0251 --                           LISTING, MONITORS COMMENTS AND FLAGS;
0251 5-                       BEGIN
0252 --
0252 --
0252 --                       PROCEDURE SET_FLAGS (STRING(1) VALUE A);
0253 --                           COMMENT ALTER THE APPROPRIATE FLAG;
0253 --                           IF     A="S" THEN FLAGS := ¬FLAGS
0253 --                           ELSE IF A="L" THEN LIST:=¬LIST
0254 --                           ELSE IF A="T" THEN TRANS:=¬TRANS
0255 --                           ELSE IF A="A" THEN ARITH:=¬ARITH
0256 --                           ELSE WRITE("INVALID FLAG CALLED ::",A);
0258 --
0258 --                       PROCEDURE GETCARD;
0259 --                           COMMENT READ AND LIST  A DATA CARD;
0259 6-                           BEGIN
0260 --                           READCARD(BUF);
0261 --                           IF BUF(0|1) = "$" THEN
0261 7-                               BEGIN
0262 --                               SET_FLAGS(BUF(1|1));
0263 --                               IF FLAGS THEN WRITE(BUF);
0264 -7                               END
0264 --                           ELSE
0265 7-                               BEGIN INTFIELDSIZE := 4;
0267 --                               LINE_NO := LINE_NO + 1;
0268 --                               IF LIST THEN WRITE(LINE_NO,"    ",BUF);
0269 --                               INTFIELDSIZE := 14;
0270 -7                               END;
0271 -6                           END GETCARD;
0272 --
0272 --                       GETCARD;
0273 --                       WHILE (BUF(0|1)="*") OR (BUF(0|1)="$") DO
0273 --                           GETCARD;
0274 --                       IBP:=0
0274 -5                       END READBUF;
0275 --
0275 --                   IBP:=IBP+1;
0276 --                   IF IBP >= 80 THEN READBUF;
0277 -4                   END BUMP_IBP;
0278 --
0278 --               PROCEDURE SKIP_COMMENT;
0279 --                   COMMENT SKIP OVER COMMENTS IN INPUT;
0279 4-                   BEGIN
0280 --                   BUMP_IBP;
0281 --                   WHILE BUF(IBP|1) ¬= "'" DO BUMP_IBP;
0282 --                   BUMP_IBP;
0283 --                   C:=BUF(IBP|1)
0283 -4                   END SKIP_COMMENT;
0284 --
0284 --               IF IBP>=80 THEN BUMP_IBP;
0285 --               C:=BUF(IBP|1);
0286 --               IF C="'" THEN SKIP_COMMENT;
0287 --               BUMP_IBP;
0288 --               NC:=BUF(IBP|1)
0288 -3               END GNC;
0289 --
0289 --           PROCEDURE LOOK_UP;
0290 --               COMMENT DETERMINE IF TOK HAS ALREADY BEEN
0290 --                   STORED.  IF NOT, CREATE ATOM HEADER CELL
0290 --                   AND PNAME ATTRIBUTE-VALUE PAIR.  RETURN POINTER
0290 --                   TO ATOM HEADER CELL;
0290 3-               BEGIN
0291 --               INTEGER ADDR;
0292 --
```

```
0292 --              LOGICAL PROCEDURE STORED (INTEGER VALUE H);
0293 --                  COMMENT DETERMINE IF ATOM HEADER CELL EXISTS--
0293 --                      IF SO, ASSIGN HEADER;
0293 4-                  BEGIN
0294 --                  STRING(72) WORD;
0295 --                  INTEGER LENGTH1;
0296 --                  LOGICAL FLAG;
0297 --                  IF H = NIL THEN FLAG := FALSE
0297 --                  ELSE
0298 5-                      BEGIN
0299 --                      GET_PNAME(CAR(H), WORD, LENGTH1);
0300 --                      IF TOK = WORD THEN
0300 6-                          BEGIN
0301 --                          HEADER := CAR(H);
0302 --                          FLAG := TRUE
0302 -6                          END
0302 --                      ELSE FLAG := STORED(CDR(H));
0304 -5                      END;
0305 --                  FLAG
0305 -4                  END STORED;
0306 --
0306 --              ADDR := HASH(LENGTH,TOK);
0307 --              IF ¬(STORED(CDR(ADDR)) OR STORED(CAR(ADDR))) THEN
0307 4-                  BEGIN
0308 --                  HEADER:=ALLOCATE;
0309 --                  BUILD_ATOM(TOK, LENGTH, HEADER);
0310 --                  PUSH(HEADER, ADDR);
0311 -4                  END
0311 -3              END LOOK_UP;
0312 --
0312 --
0312 --          PROCEDURE BUILD_TOK;
0313 --              COMMENT ADD NEXT CHARACTER TO TOK;
0313 3-              BEGIN
0314 --              IF LENGTH >= 72 THEN SYN_ERR(1);
0315 --              TOK(LENGTH+1) := C;
0316 --              LENGTH:=LENGTH+1;
0317 -3              END BUILD_TOK;
0318 --
0318 --          COMMENT MAIN OF SCANNER;
0318 --          LENGTH := 0;
0319 --          TOK := " ";
0320 --          IF C<"A" THEN
0320 3-              BEGIN
0321 --              IF C="""" THEN
0321 4-                  BEGIN
0322 --                  WHILE NC¬="""" DO
0322 5-                      BEGIN
0323 --                      GNC;
0324 --                      BUILD_TOK;
0325 -5                      END;
0326 --                  GNC;
0327 --                  TYPE := CONSTANT;  LOOK_UP;
0328 -4                  END
0329 --              ELSE IF ((C="#") OR (C="<") OR (C=">") OR (C="$") OR
0330 --                       (C="&") OR (C="(") OR (C=")") OR (C=";") OR
0330 --                       (C="¬") OR (C=" ") OR (C=".")) THEN
0330 4-                  BEGIN
0331 --                  TOK := C;  TYPE := SPECIAL;
0333 -4                  END
0333 --              ELSE
0334 4-                  BEGIN
0335 --                  TOK := C;  TYPE := IDENTIFIER;
0337 --                  LENGTH := 1;   LOOK_UP
0338 -4                  END
0338 -3              END
0338 --          ELSE
0339 --          IF (C>="A") AND (C<="Z") THEN
0339 3-              BEGIN
0340 --              WHILE NC >= "A" DO
0340 4-                  BEGIN
0341 --                  BUILD_TOK;
0342 --                  GNC;
0343 -4                  END;
0344 --              BUILD_TOK;
0345 --              TYPE:=IDENTIFIER;
0346 --              LOOK_UP;
0347 -3              END
0347 --          ELSE
0348 --          IF C>="0" THEN
0348 3-              BEGIN INTEGER SUM;
0350 --              SUM := 0;
0351 --              WHILE NC >= "0" DO
0351 4-                  BEGIN
0352 --                  SUM := 10*SUM + (DECODE(C) - 240);
0353 --                  IF SUM > (MAXINTEGER DIV 10) THEN SYN_ERR(2);
0354 --                  GNC;
0355 -4                  END;
0356 --              SUM := 10*SUM + (DECODE(C) - 240);
0357 --              TYPE:=NUMS;
0358 --              HEADER := BUILD_C_ATOM(SUM);
0359 -3              END
0359 --          ELSE
0360 --          SYN_ERR(3);
0361 --          GNCT
0362 --          WHILE C=" " DO GNC;
0363 -2          END SCANNER;
```

```
0364 --
0364 --      COMMENT**********************************************************
0364 --      *                                                                *
0364 --      *      SECTION VIII:  TRANSLATION                                *
0364 --      *                                                                *
0364 --      ******************************************************************;
0364 --
0364 --      INTEGER PROCEDURE BUILDS;
0365 --          COMMENT LEFT PARENTHESIS ALREADY SEEN;
0365 2-          BEGIN
0366 --          INTEGER T;
0367 --          SCANNER;
0368 --          IF TOK = ")" THEN NIL
0368 --          ELSE IF TOK = "." THEN
0369 3-              BEGIN SCANNER;
0371 --              IF TOK = "(" THEN T := BUILDS
0371 --              ELSE T := HEADER;
0373 --              SCANNER;
0374 --              IF TOK ¬= ")" THEN SYN_ERR(7);
0375 --              T
0375 -3              END
0375 --          ELSE
0376 3-              BEGIN
0377 --              IF (TYPE = IDENTIFIER) OR (TYPE = NUMS) THEN
0377 --                  T := HEADER
0377 --              ELSE IF TOK = "(" THEN T := BUILDS
0378 --              ELSE SYN_ERR(7);
0380 --              CONS(T,BUILDS)
0380 -3              END
0380 -2          END BUILDS;
0381 --
0381 --      INTEGER PROCEDURE TFUNC;
0382 --          COMMENT TRANSLATE AN M-EXPRESSION FUNCTION
0382 --              INTO AN INTERNAL S-EXPRESSION;
0382 2-          BEGIN
0383 --
0383 --          INTEGER PROCEDURE LABEL_FUNC;
0384 --              COMMENT TRANSLATE A-LABEL FUNCTION;
0384 3-              BEGIN
0385 --              INTEGER FF,FUNC;
0386 --              SCANNER;
0387 --              IF TOK¬= "<" THEN SYN_ERR(4);
0388 --              SCANNER;
0389 --              FF := HEADER;
0390 --              SCANNER;
0391 --              IF TOK ¬= ";" THEN SYN_ERR(4);
0392 --              SCANNER;
0393 --              FUNC := TFUNC;
0394 --              SCANNER;
0395 --              IF TOK ¬= ">" THEN SYN_ERR(4);
0396 --              CONS3(LABEL,FF,FUNC)
0396 -3              END LABEL_FUNC;
0397 --
0397 --
0397 --          INTEGER PROCEDURE LAMBDA_FUNC;
0398 --              COMMENT TRANSLATE A LAMBDA FUNCTION;
0398 3-              BEGIN
0399 --
0399 --              INTEGER PROCEDURE VARLIST;
0400 4-                  BEGIN INTEGER T; SCANNER;
0403 --                  IF TOK = ">"  THEN
0403 --                      T := NIL
0403 --                  ELSE IF (TOK="<") OR (TOK=";") THEN
0404 5-                      BEGIN SCANNER;
0406 --                      T := CONS(HEADER,VARLIST);
0407 -5                      END
0407 --                  ELSE SYN_ERR(10);
0409 --                  T
0409 -4                  END VARLIST;
0410 --
0410 --              INTEGER VLIST,FORM;
0411 --              SCANNER;
0412 --              IF TOK¬="<" THEN SYN_ERR(5);
0413 --              VLIST := VARLIST;
0414 --              SCANNER;
0415 --              IF TOK¬=";" THEN SYN_ERR(5);
0416 --              FORM := TFORM;
0417 --              SCANNER;
0418 --              IF TOK ¬= ">" THEN SYN_ERR(5);
0419 --              CONS3(LAMBDA,VLIST,FORM)
0419 -3              END LAMBDA_FUNC;
0420 --
0420 --
0420 --          INTEGER FN;
0421 --          IF TYPE = IDENTIFIER THEN FN := HEADER
0421 --          ELSE IF TOK="$"      THEN FN := LAMBDA_FUNC
0422 --          ELSE IF TOK="@"      THEN FN := LABEL_FUNC
0423 --          ELSE SYN_ERR(6);
0425 --          FN
0425 -2          END TFUNC;
0426 --
```

```
0426 --        INTEGER PROCEDURE TFORM;
0427 --            COMMENT TRANSLATE A  FORM;
0427 2-            BEGIN
0428 --
0428 --            INTEGER PROCEDURE ARG_LIST;
0429 --                COMMENT CONSTRUCT AN ARGUMENT LIST;
0429 3-                BEGIN
0430 --                INTEGER T;
0431 --                SCANNER;
0432 --                IF TOK = ">" THEN T := NIL
0432 --                ELSE IF (TOK ="<")OR(TOK=";") THEN
0433 --                    T := CONS(TFORM,ARG_LIST)
0433 --                ELSE SYN_ERR(9);
0435 --                T
0435 -3               END ARG_LIST;
0436 --
0436 --            INTEGER PROCEDURE BUILD_COND;
0437 --                COMMENT TRANSLATE CONDITIONAL M-EXPRESSION INTO
0437 --                    INTERNAL S-EXPRESSION;
0437 3-                BEGIN
0438 --
0438 --                INTEGER PROCEDURE BUILD_P_E_PAIRLIST;
0439 --                    COMMENT BUILD PREDICATE-EXPRESSION PAIR LIST;
0439 4-                    BEGIN
0440 --                    INTEGER T2;
0441 --
0441 --                    INTEGER PROCEDURE P_E_PAIR;
0442 --                        COMMENT BUILD PREDICATE-EXPRESSION PAIR;
0442 5-                        BEGIN
0443 --                        INTEGER P,E;
0444 --                        P := TFORM;
0445 --                        SCANNER;
0446 --                        IF TOK = "¬" THEN
0446 --                            P := CONS(TFORM,NIL)
0446 --                        ELSE SYN_ERR(8);
0448 --                        CONS(P,E)
0448 -5                       END P_E_PAIR;
0449 --
0449 --                    IF TOK=">" THEN T2:=NIL
0449 --                    ELSE IF (TOK = "<") OR (TOK = ";") THEN
0450 5-                        BEGIN
0451 --                        T2:=P_E_PAIR;  SCANNER;
0453 --                        T2:=CONS(T2,BUILD_P_E_PAIRLIST)
0453 -5                       END
0453 --                   ELSE SYN_ERR(8);
0455 --                    T2
0455 -4                   END P_E_PAIRLIST;
0456 --
0456 --                CONS(COND,BUILD_P_E_PAIRLIST)
0456 -3               END BUILD_COND;
0457 --
0457 --            COMMENT MAIN OF TFORM;
0457 --            SCANNER;
0458 --
0458 --            IF (TYPE=CONSTANT) OR (TYPE=NUMS) THEN
0458 --                COMMENT FORM IS A CONSTANT;
0458 --                CONS(QUOTE, CONS(HEADER, NIL))
0458 --
0458 --            ELSE IF TOK = "(" THEN
0459 --                COMMENT FORM IS AN S-EXPRESSION;
0459 --                CONS(QUOTE,CONS(BUILDS,NIL))
0459 --
0459 --            ELSE IF (TYPE=IDENTIFIER) AND ( C¬="<") THEN
0460 --                COMMENT FORM IS A VARIABLE;
0460 --                HEADER
0460 --
0460 --            ELSE IF TOK = "<" THEN
0461 --                COMMENT FORM IS A CONDITIONAL;
0461 --                BUILD_COND
0461 --
0461 --            ELSE COMMENT FORM IS A FUNCTION<ARG LIST> OR A FUNCTION
0462 --            (FUNCTIONAL ARGUMENT);
0462 3-                BEGIN INTEGER T;
0464 --                T := TFUNC;
0465 --                IF C = "<" THEN CONS(T,ARG_LIST)
0465 --                ELSE CONS(QUOTE, CONS(T, NIL))
0466 -3                END
0466 --
0466 -2           END TFORM;
0467 --
0467 --
0467 --        INTEGER PROCEDURE TARGS;
0468 --            COMMENT BUILD ARGUMENT LIST OF S-EXPRESSIONS;
0468 2-            BEGIN
0469 --            INTEGER T;
0470 --            SCANNER;
0471 --            IF TOK = ">" THEN T := NIL
0471 --            ELSE IF (TOK ="<") OR (TOK=";") THEN
0472 3-                BEGIN SCANNER;
0474 --                IF (TYPE=IDENTIFIER) OR (TYPE=NUMS) THEN
0474 --                    T := CONS(HEADER,TARGS)
0474 --                ELSE IF TOK = "(" THEN
0475 --                    T := CONS(BUILDS,TARGS)
0475 --                ELSE SYN_ERR(9);
0477 -3                END;
0478 --                T
0478 -2           END TARGS;
```

```
0479 --       COMMENT************************************************************
0479 --       *                                                                   *
0479 --       *      SECTION IX:  INTERPRETATION                                  *
0479 --       *                                                                   *
0479 --       *********************************************************************;
0479 --
0479 --       INTEGER PROCEDURE EVALQUOTE(INTEGER VALUE FN, X);
0480 --           COMMENT EVALUATE THE FUNCTION AT 'FN' APPLIED TO
0480 --           THE ARGUMENTS AT 'X';
0480 2-           BEGIN
0481 --
0481 --           INTEGER PROCEDURE COMPOSE(INTEGER VALUE X,Y);
0482 --               COMMENT RETURN DATA ELEMENT AND ASSOCIATED
0482 --                   G-VECTOR REPRESENTING FUNCTIONAL COMPOSITION;
0482 --               CONS(CAR(X),APPEND(Y,CDR(X)));
0483 --
0483 --           INTEGER PROCEDURE APPEND(INTEGER VALUE X,Y);
0484 --               COMMENT APPEND LIST Y TO LIST X;
0484 --               IF X = NIL THEN Y
0484 --               ELSE
0485 3-                   BEGIN INTEGER T;
0487 --                   T := X;
0488 --                   WHILE CDR(T) =/= NIL DO
0488 --                       T := CDR(T);
0489 --                   SETCDR(T,Y);
0490 --                   X
0490 -3                  END;
0491 --
0491 --           INTEGER PROCEDURE APPLY(INTEGER VALUE FN, X, A);
0492 --               COMMENT APPLY THE FUNCTION TO ITS ARGUMENTS AND
0492 --               GENERATE THE ASSOCIATION LIST;
0492 3-               BEGIN
0493 --
0493 --               INTEGER PROCEDURE INCR_G;
0494 --                   IF ARITH AND (FN <=IMUL) THEN
0494 --                       CDR(X)
0494 --                   ELSE
0495 --                       APPEND(CDR(X),CONS(BUILD_C_ATOM(1),NIL));
0496 --
0496 --               INTEGER PROCEDURE PAIRLIS(INTEGER VALUE X, Y, A);
0497 --                   COMMENT BUILD A LIST OF PAIRS OF CORRESPONDING
0497 --                   ELEMENTS OF LISTS X AND Y AND APPEND THIS NEW LIST
0497 --                   TO THE ASSOCIATION LIST;
0497 --                   IF NUL(X) AND NUL(Y) THEN
0497 --                       A
0497 --                   ELSE IF ATOM(X) OR ATOM(Y) THEN
0498 --                       EXEC_ERR(3,A)
0498 --                   ELSE
0499 --                       CONS(CONS(CAR(X),CAR(Y)),
0499 --                                   PAIRLIS(CDR(X),CDR(Y),A));
0500 --
0500 --               COMMENT MAIN OF APPLY;
0500 --               IF ATOM(CAR(X)) THEN
0500 --                   EXEC_ERR(5,CAR(X))
0500 --               ELSE IF ATOM(FN) THEN
0501 4-                   BEGIN
0502 --                   IF EQ(FN,ICAR) THEN
0502 5-                       BEGIN
0503 --                       IF ATOM(CAAR(X)) THEN
0503 --                           EXEC_ERR(4,CAAR(X))
0503 --                       ELSE
0504 --                           CONS(CAAR(CAR(X)), INCR_G)
0504 -5                      END
0504 --                   ELSE IF EQ(FN,ICDR) THEN
0505 --                       CONS(CDAR(CAR(X)), INCR_G)
0505 --                   ELSE IF EQ(FN,ICONS) THEN
0506 --                       CONS(CONS(CAAR(X),CADAR(X)), INCR_G)
0506 --                   ELSE IF EQ(FN,IATOM) THEN
0507 5-                       BEGIN
0508 --                       IF ATOM(CAAR(X)) THEN CONS(T, INCR_G)
0508 --                       ELSE CONS(F, INCR_G)
0509 -5                      END
0509 --                   ELSE IF EQ(FN,IEQ) THEN
0510 5-                       BEGIN
0511 --                       IF EQ (CAAR(X), CADAR(X)) THEN
0511 --                           CONS(T, INCR_G)
0511 --                       ELSE CONS(F, INCR_G)
0512 -5                      END
0512 --                   ELSE IF EQ(FN, IADD) THEN
0513 --                       CONS(ADD(CAAR(X),CADAR(X)), INCR_G)
0514 --                   ELSE IF EQ(FN, IMUL) THEN
0514 --                       CONS(MUL(CAAR(X),CADAR(X)), INCR_G)
0514 --                   ELSE
0515 --                       APPLY(CAR(EVAL(FN,A)),X,A)
0515 -4                  END
0515 --                   ELSE IF EQ(CAR(FN),LAMBDA) THEN
0516 4-                   BEGIN
0516 --                   INTEGER P, TEMP;
0516 --                   P := PAIRLIS(CADR(FN),CAR(X),A);
0517 --                   TEMP:=EVAL(CADDR(FN), P);
0517 --                   FREE_A_LIST(P,A);
0517 --                   COMPOSE(TEMP, CDR(X))
0517 -4                  END
0517 --                   ELSE IF EQ(CAR(FN),LABEL) THEN
0518 --                       APPLY(CADDR(FN),X,
0518 --                                   CONS(CONS(CADR(FN),CADDR(FN)),A))
```

```
0522 --              ELSE EXEC_ERR(1,FN)
0523 -3              END APPLY;
0524 --
0524 --          INTEGER PROCEDURE EVAL(INTEGER VALUE E, A);
0525 --              COMMENT EVAL HANDLES FORMS;
0525 3-              BEGIN
0526 --
0526 --              INTEGER PROCEDURE ASSOC(INTEGER VALUE X,A);
0527 --                  COMMENT RETURN THE FIRST PAIR IN THE ASSOCIATION
0527 --                  LIST WHOSE FIRST TERM IS X;
0527 --                  IF NUL (A) THEN
0527 --                      EXEC_ERR(2,X)
0527 --                  ELSE IF EQ(CAAR(A),X) THEN
0528 --                      CAR(A)
0528 --                  ELSE
0529 --                      ASSOC(X,CDR(A));
0530 --
0530 --
0530 --              INTEGER PROCEDURE SUM(INTEGER VALUE X,Y);
0531 --                  COMMENT SUM CORRESPONDING ELEMENTS OF X AND Y;
0531 4-                  BEGIN INTEGER T;
0533 --                  IF NUL(X) THEN T := Y
0533 --                  ELSE IF NUL(Y) THEN T := X
0534 --                  ELSE
0535 5-                      BEGIN
0536 --                      T := CONS(ADD(CAR(X),CAR(Y)),
0536 --                                SUM(CDR(X),CDR(Y)));
0537 --                      FREE(CDAR(X)); FREE(CAR(X)); FREE(X);
0540 --                      FREE(CDAR(Y)); FREE(CAR(Y)); FREE(Y);
0543 -5                      END;
0544 --                  T
0544 -4                  END SUM;
0545 --
0545 --              INTEGER PROCEDURE COMBINE(INTEGER VALUE X,Y);
0546 --                  COMMENT COMBINE EVAULUATED ARGUMENTS AND
0546 --                      G-VECTORS;
0546 --                  IF NUL(Y) THEN
0546 --                      CONS(CONS(CAR(X),NIL),CDR(X))
0546 --                  ELSE
0547 --                      CONS(CONS(CAR(X),CAR(Y)), SUM(CDR(X),CDR(Y)));
0548 --
0548 --              INTEGER PROCEDURE EVLIS(INTEGER VALUE M,A);
0549 --                  COMMENT RETURN EVALUATED ARGUMENT LIST
0549 --                      AND COMBINED G-VECTOR;
0549 4-                  BEGIN
0550 --                  IF NUL(M) THEN
0550 --                      NIL
0550 --                  ELSE
0551 --                      COMBINE(EVAL(CAR(M),A),EVLIS(CDR(M),A))
0551 -4                  END EVLIS;
0552 --
0552 --              INTEGER PROCEDURE EVCON(INTEGER VALUE C,A);
0553 --                  COMMENT RETURN EVALUATED CONDITIONAL AND
0553 --                      G-VECTOR;
0553 4-                  BEGIN
0554 --                  IF C = NIL THEN
0554 --                      EXEC_ERR(6,A)
0554 --                  ELSE
0555 --                      GRAPHCON(EVAL(CAAR(C),A),C,A)
0555 -4                  END EVCON;
0556 --
0556 --              INTEGER PROCEDURE GRAPHCON(INTEGER VALUE P,C,A);
0557 --                  COMMENT COMPOSE G-VECTOR FROM PREDICATE WITH
0557 --                      G-VECTOR FROM REMAINDER OF CONDITIONAL AND
0557 --                      RETURN WITH VALUE OF CONDITIONAL;
0557 --                  IF CAR(P) = T THEN
0557 --                      COMPOSE(EVAL(CADAR(C),A),CDR(P))
0557 --                  ELSE IF CAR(P) = F THEN
0558 --                      COMPOSE(EVCON(CDR(C),A),CDR(P))
0558 --                  ELSE EXEC_ERR(7,CAAR(C));
0560 --
0560 --              COMMENT MAIN OF EVAL;
0560 --              IF ATOM(E) THEN
0560 --                  CONS(CDR(ASSOC(E,A)),NIL)
0560 --              ELSE IF ATOM(CAR(E)) THEN
0561 4-                  BEGIN
0562 --                  IF EQ(CAR(E),QUOTE) THEN
0562 --                      CONS(CADR(E),NIL)
0562 --                  ELSE IF EQ(CAR(E),COND) THEN
0563 --                      EVCON(CDR(E),A)
0563 --                  ELSE
0564 --                      APPLY(CAR(E),EVLIS(CDR(E),A),A)
0564 -4                  END
0564 --              ELSE
0565 --                  APPLY(CAR(E),EVLIS(CDR(E),A),A)
0565 -3              END EVAL;
0566 --
0566 --          COMMENT MAIN OF EVALQUOTE;
0566 --
0566 --          APPLY(FN,CONS(X,NIL),NIL)
0566 --
0566 -2          END EVALQUOTE;
0567 --
```

```
0567 --       COMMENT**********************************************************
0567 --       *                                                                *
0567 --       *      SECTION X:  ERROR HANDLING                                *
0567 --       *                                                                *
0567 --       ******************************************************************;
0567 --
0567 --       PROCEDURE ERROR(INTEGER VALUE K);
0568 --           COMMENT HANDLE PROGRAMMING ERRORS;
0568 2-           BEGIN
0569 --           CASE K OF
0569 3-               BEGIN
0570 --               WRITE("ATTRIBUTE NOT ON PROPERTY LIST");
0571 --               WRITE("PROGRAM TOO LARGE TO EXECUTE IN ",
0571 --                     "PRESENT MEMORY SPACE");
0572 --               WRITE("ATTEMPTED STORAGE OF WORD OF LENGTH ZERO");
0573 -3               END;
0574 --           RECOVER;
0575 -2           END;
0576 --
0576 --       PROCEDURE SYN_ERR (INTEGER VALUE K);
0577 --           COMMENT HANDLE SYNTAX ERRORS;
0577 2-           BEGIN
0578 --           STRING(80) ERRBUF;
0579 --           IF ¬LIST THEN WRITE("         ",BUF);
0580 --           ERRBUF := " ";
0581 --           ERRBUF[ISP+1] := "|";
0582 --           WRITE("       ISP:",ERRBUF);
0583 --           WRITE("****** ");
0584 --           WRITEON("SYNTAX ERROR FOUND ON LINE_NO",LINE_NO);
0585 --           WRITEON(" ******");
0586 --           WRITE(" ");
0587 --           CASE K OF
0587 3-               BEGIN
0588 --               WRITE ("TOKEN LENGTH EXCEEDS 72");
0589 --               WRITE ("INTEGER VALUE TOO LARGE");
0590 --               WRITE ("ILLEGAL CHARACTER");
0591 --               WRITE ("IMPROPER LABEL CONSTRUCT");
0592 --               WRITE ("IMPROPER LAMBDA EXPRESSION");
0593 --               WRITE ("UNDECODABLE M-EXPRESSION");
0594 --               WRITE ("IMPROPERLY PARENTHESIZED EXPRESSION");
0595 --               WRITE ("INVALID CONDITIONAL EXPRESSION");
0596 --               WRITE ("INVALID ARGUMENT LIST CONSTRUCTION");
0597 --               WRITE ("INVALID VARIABLE LIST");
0598 -3               END;
0599 --           RECOVER
0599 -2           END SYN_ERR;
0600 --
0600 --       INTEGER PROCEDURE EXEC_ERR (INTEGER VALUE X, Y);
0601 --           COMMENT HANDLE SEMANTIC ERRORS;
0601 2-           BEGIN
0602 --           CASE X OF
0602 3-               BEGIN
0603 --               WRITE("ERROR- IMPROPERLY DEFINED FUNCTION:");
0604 --               WRITE("ERROR- UNDEFINED VARIABLE OR FUNCTION:");
0605 --               WRITE("ERROR- VARIABLE LIST DOES NOT MATCH ",
0605 --                     "ARGUMENT LIST. LATEST A-LIST IS:");
0606 --               WRITE("ERROR- ATTEMPTED CAR OF ATOM. ");
0607 --               WRITE("ERROR- ARGUMENT LIST CANNOT BE AN ATOM. ",
0607 --                     "ARGUMENT LIST IS:");
0608 --               WRITE("ERROR- UNDEFINED CONDITIONAL.  ",
0608 --                     "LATEST A-LIST IS:");
0609 --               WRITE("PREDICATE DOES NOT GIVE TRUTH VALUE.",
0609 --                     "  PREDICATE IS:");
0610 -3               END;
0611 --           OUTPUT (Y);
0612 --           RECOVER;
0613 --           X
0613 -2           END EXEC_ERR;
0614 --
0614 --       PROCEDURE RECOVER;
0615 --           COMMENT RECOVER FROM FATAL ERRORS;
0615 2-           BEGIN
0616 --           IF INTRANS THEN
0616 3-               BEGIN INTFIELDSIZE := 4;
0618 --               WHILE BUF[0[1] <= "*" DO
0618 4-                   BEGIN
0619 --                   READCARD(BUF);  INTFIELDSIZE:=4;
0621 --                   LINE_NO := LINE_NO+1;  ISP:=0;
0623 --                   IF LIST THEN
0623 --                       WRITE(LINE_NO,"   ",BUF);
0624 -4                   END;
0625 --               INTFIELDSIZE := 14;
0626 -3               END;
0627 --           GO TO FINIS;
0628 -2           END RECOVER;
0629 --
```

```
0629 --       COMMENT**********************************************************
0629 --       *                                                                *
0629 --       *      SECTION XI:  OUTPUT                                       *
0629 --       *                                                                *
0629 --       ******************************************************************;
0629 --
0629 --       PROCEDURE OUTPUT(INTEGER VALUE X);
0630 --           COMMENT PRINT LIST ROOTED AT X;
0630 2-           BEGIN
0631 --           STRING(72) OUTBUF;
0632 --           INTEGER OBP;
0633 --
0633 --           PROCEDURE DUMP;
0634 --               COMMENT PRINT OUTPUT BUFFER;
0634 3-               BEGIN
0635 --               WRITE(OUTBUF); SKIP(1);
0637 --               OBP := 0;  OUTBUF := " ";
0638 -3               END DUMP;
0639 --
0639 --           PROCEDURE BUMP_OBP;
0640 --               COMMENT MANAGES OUTPUT BUFFER SIZE;
0640 3-               BEGIN
0641 --               OBP := OBP+1;
0642 --               IF OBP >= 72 THEN DUMP;
0643 -3               END BUMP_OBP;
0644 --
0644 --           PROCEDURE PRINT_ATOM(INTEGER VALUE X);
0645 --               COMMENT PUT ATOM INTO OUTPUT BUFFER;
0645 3-               BEGIN
0646 --               STRING(72) BUFF;
0647 --               INTEGER LENGTH;
0648 --
0648 --               LOGICAL PROCEDURE NUM_ATOM(INTEGER VALUE X);
0649 --                   COMMENT DETERMINE IF AN ATOM HEADER CELL
0649 --                       POINTS TO A NUMBER;
0649 --                   BITSTRING(CAR(X)) = #0000FFFF;
0650 --
0650 --               PROCEDURE DUMP_NUM (INTEGER VALUE N);
0651 --                   COMMENT DUMP NUMBER TO OUTBUF;
0651 4-                   BEGIN LENGTH := LENGTH + 1;
0653 --                   IF (LENGTH + OBP) > 72 THEN DUMP;
0654 --                   IF N > 9 THEN DUMP_NUM(N DIV 10);
0655 --                   OUTBUF(OBP|1) := CODE((N REM 10) + 240);
0656 --                   BUMP_OBP;
0657 -4                   END DUMP_NUM;
0658 --
0658 --               IF NUM_ATOM(X) THEN
0658 4-                   BEGIN LENGTH := 0;
0660 --                   DUMP_NUM(NUMBER(M(CDR(X))));
0661 -4                   END
0661 --               ELSE
0662 4-                   BEGIN
0663 --                   GET_PNAME(X,BUFF,LENGTH);
0664 --                   IF LENGTH > (71-OBP) THEN  DUMP;
0665 --                   FOR I:=0 UNTIL LENGTH-1 DO
0665 5-                       BEGIN
0666 --                       OUTBUF(OBP|1) := BUFF(I|1);
0667 --                       BUMP_OBP;
0668 -5                       END;
0669 -4                   END;
0670 -3               END PRINT_ATOM;
0671 --
0671 --           PROCEDURE WRITE_S(INTEGER VALUE X);
0672 --               COMMENT WRITE S-EXPRESSION AT X ONTO OUTPUT BUFFER;
0672 3-               BEGIN
0673 --               IF ATOM(X) THEN
0673 --                   PRINT_ATOM(X)
0673 --               ELSE
0674 4-                   BEGIN
0675 --                   OUTBUF(OBP|1) := "(";  BUMP_OBP;
0677 --                   WRITE_S(CAR(X));
0678 --                   X := CDR(X);
0679 --                   WHILE X ¬= NIL DO
0679 --                       IF ATOM(X) THEN
0679 5-                           BEGIN
0680 --                           OUTBUF(OBP|1) := ".";  BUMP_OBP;
0682 --                           PRINT_ATOM(X);   X:=NIL
0683 -5                           END
0683 --                       ELSE
0684 5-                           BEGIN
0685 --                           BUMP_OBP;  COMMENT OUTPUT A BLANK;
0686 --                           WRITE_S(CAR(X));   X:=CDR(X)
0687 -5                           END;
0688 --                   OUTBUF(OBP|1) := ")";  BUMP_OBP;
0690 -4                   END;
0691 -3               END WRITE_S;
0692 --
0692 --
0692 --           OBP := 0;
0693 --           OUTBUF := " ";
0694 --           WRITE_S(X);
0695 --           DUMP;
0696 -2           END OUTPUT;
0697 --
0697 --
```

```
0697 --       PROCEDURE SKIP(INTEGER VALUE X);
0698 --           COMMENT SKIP X NUMBER OF LINES;
0698 --           FOR I:=1 UNTIL X DO WRITE(" ");
0699 --
0699 --       COMMENT**************************************************
0699 --       *                                                        *
0699 --       *      SECTION X:  MONITOR                               *
0699 --       *                                                        *
0699 --       **************************************************;
0699 --
0699 --
0699 --       PROCEDURE MONITOR;
0700 --           COMMENT INVOKE TRANSLATION ROUTINES, OUTPUT TRANSLATION
0700 --               IF TRANS IS TRUE, INVOKE INTERPRETER ROUTINES AND
0700 --               OUTPUT RESULTS;
0700 --
0700 2-            BEGIN
0701 --            INTEGER FN, ARGS, VAL;
0702 --            INTEGER MAXWIDTH, MAXPATHLENGTH, NODESUM;
0703 --            REAL SPEEDUPRATIO;
0704 --
0704 --            PROCEDURE GETSTATS (INTEGER VALUE VECTOR);
0705 3-                BEGIN
0706 --                MAXWIDTH := 0;  MAXPATHLENGTH := 0;
0708 --                NODESUM := 0;
0709 --                WHILE ¬NUL(VECTOR) DO
0709 4-                    BEGIN INTEGER WIDTH;
0711 --                    WIDTH := NUMBER(M(CDAR(VECTOR)));
0712 --                    NODESUM := NODESUM + WIDTH;
0713 --                    IF WIDTH > MAXWIDTH THEN
0713 --                        MAXWIDTH := WIDTH;
0714 --                    MAXPATHLENGTH := MAXPATHLENGTH + 1;
0715 --                    VECTOR := CDR(VECTOR);
0716 -4                    END;
0717 --                SPEEDUPRATIO := NODESUM / MAXPATHLENGTH;
0718 -3                END GETSTATS;
0719 --
0719 --            INTRANS := TRUE;
0720 --            SCANNER;
0721 --            FN := TFUNC;
0722 --            ARGS := TARGS;
0723 --            INTRANS := FALSE;
0724 --            IF TRANS THEN
0724 3-                BEGIN
0725 --                SKIP(2);
0726 --                WRITE("****** TRANSLATION FOLLOWS ******");
0727 --                SKIP(1);
0728 --                OUTPUT(FN);
0729 --                OUTPUT(ARGS);
0730 -3                END;
0731 --            SKIP(2);
0732 --            WRITE("****** EVALUATION BEGINS ******");
0733 --            VAL := EVALQUOTE(FN, ARGS);
0734 --            SKIP(1);
0735 --            WRITE("RESULT IS:");
0736 --            OUTPUT(VAL);
0737 --            SKIP(2);
0738 --            GETSTATS(CDR(VAL));
0739 --            WRITE("PROCESSORS REQUIRED FOR OPTIMUM PARALLELING.",
0739 --                MAXWIDTH);
0740 --            WRITE("EXECUTION STEPS (PARALLEL).....",
0740 --                MAXPATHLENGTH);
0741 --            WRITE("EXECUTION STEPS (SEQUENTIAL).....",
0741 --                NODESUM);
0742 --            WRITE("SPEED-UP RATIO (SEQUENTIAL/PARALLEL).....",
0742 --                SPEEDUPRATIO);
0743 -2            END MONITOR;
0744 --
0744 --
0744 --
0744 --       COMMENT**************************************
0744 --       *                                            *
0744 --       *              MAIN PROGRAM                  *
0744 --       *                                            *
0744 --       **************************************;
0744 --
0744 --       INITIALIZE;  MONITOR;
0746 --
0746 --       FINIS :
0746 --           WRITE("FREE STORAGE REMAINING:",FREE_CELLS);
0747 --
0747 -1       END.

EXECUTION OPTIONS: DEBUG,1  TIME=10 SECONDS PAGES=20 MARGIN=72

008.33 SECONDS IN COMPILATION, (40748, 07328) BYTES OF CODE GENERATED
```

LIST OF REFERENCES

1. Thurber, K. J. and Wald, L. D., "Associative and Parallel Processors," Computing Surveys, v. 7, no. 4, p. 215-255, December 1975.

2. Glushkov, V. M., and others, "Recursive Machines and Computing Technology," Information Processing 74, p. 65-73, North-Holland Publishing Company, 1974.

3. Barnes, G. H., and others, "The ILLIAC IV Computer," IEEE Transactions on Computers, v. C-18. no. 8, p. 746-757, August 1968.

4. Kuck, D. J., "ILLIAC IV Software and Application Programming," IEEE Transactions on Computers, v. C-17, no. 8, August 1968.

5. Ramamoorthy, C. V., and Gonzales, M. J., "A Survey of Techniques for Recognizing Parallel Processable Streams in Computer Programs," Proc. AFIPS 1969 Fall Joint Computer Conference, v. 35, AFIPS Press, Montvale, N. J., p. 1-15, 1969.

6. Lamport, Leslie, "The Parallel Execution of DO Loops," Communications of the ACM, v. 17, no. 2, p. 83-93, February 1974.

7. Keller R. M., "Look-Ahead Processors," Computing Surveys, v. 7, no. 4, p. 177-195, December 1975.

8. Stone, H. S., "Problems of Parallel Computation," Complexity of Sequential and Parallel Numerical Algorithms, Traub, J. F., ed., Academic Press, 1973.

9. Brent, R. P., "The Parallel Evaluation of Arithmetic Expressions in Logarithmic Time," Complexity of Sequential and Parallel Numerical Algorithms, Traub, J. F., ed., Academic Press, 1973.

10. McCarthy, J., and others, LISP 1.5 Programmer's Manual, 2nd ed., The M. I. T. Press, 1965.

11. McCarthy, J., "A Basis for a Mathematical Theory of Computation," Proceedings of the Western Joint Computer Conference, p. 225-237, May 1961.

12. Murtha, J. C., "Highly Parallel Information Processing Systems," Advances in Computers, Alt, F. L., and Rubinoff, M., editors, Academic Press, 1966.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Library, Code 0142 Naval Postgraduate School Monterey, California 93940	2
2.	Department Chairman, Code 72 Computer Science Department Naval Postgraduate School Monterey, California 93940	4
3.	Assoc. Professor G. A. Kildall, Code 52Kd Computer Science Department Naval Postgraduate School Monterey, California 93901	1
4.	Assoc. Professor D. L. Davis, Code 53Dv Mathematics Department Naval Postgraduate School Monterey, California 93901	1
5.	LCDR F. Burkhead, USN 22306 Capote Salinas, California 93901	2
6.	Defense Documentation Center Cameron Station Alexandria, Virginia 22314	2